ARIZONA STATE UNIVERSITY

ARIZONA STATE UNIVERSITY

GENERAL STUDIES PROGRAM COURSE PROPOSAL COVER FORM

Courses submitted to the GSC between 2/1 and 4/30 if approved, will be effective the following Spring.

Courses submitted between 5/1 and 1/31 if approved, will be effective the following Fall.

(SUBMISSION VIA ADOBE.PDF FILES IS PREFERRED)

DATE 03/01/2010

1. ACADEMIC UNIT: Geographical Sciences and Urban Planning

2. COURSE PROPOSED: GPH (prefix) 471 (number) Geographics (title) 3 (semester hours)

3. CONTACT PERSON: Name: Sergio J. Rey Phone: 619 928-4499

   Mail Code: 5301 E-Mail: srey@asu.edu

4. ELIGIBILITY: New courses must be approved by the Tempe Campus Curriculum Subcommittee and must have a regular course number. For the rules governing approval of omnibus courses, contact the General Studies Program Office at 965–0739.

5. AREA(S) PROPOSED COURSE WILL SERVE. A single course may be proposed for more than one core or awareness area. A course may satisfy a core area requirement and more than one awareness area requirements concurrently, but may not satisfy requirements in two core areas simultaneously, even if approved for those areas. With departmental consent, an approved General Studies course may be counted toward both the General Studies requirement and the major program of study. **(Please submit one designation per proposal)**

Core Areas	Awareness Areas
Literacy and Critical Inquiry–L ☐	Global Awareness–G ☐
Mathematical Studies–MA ☐ CS ☒	Historical Awareness–H ☐
Humanities, Fine Arts and Design–HU ☐	Cultural Diversity in the United States–C ☐
Social and Behavioral Sciences–SB ☐	
Natural Sciences–SQ ☐ SG ☐	

6. DOCUMENTATION REQUIRED.
   (1) Course Description
   (2) Course Syllabus
   (3) Criteria Checklist for the area
   (4) Table of Contents from the textbook used, if available

7. ***In the space provided below (or on a separate sheet), please also provide a description of how the course meets the specific criteria in the area for which the course is being proposed.***

CROSS-LISTED COURSES: ☐ No ☒ Yes; Please identify courses: ____

Is this amultisection course?: ☐ No ☒ Yes; Is it governed by a common syllabus? ____

LUC ANSELIN

Chair/Director (Print or Type)

Chair/Director (Signature) [signature]

Rev. 1/94, 4/95, 7/98, 4/00, 1/02, 10/08

ARIZONA STATE UNIVERSITY

Date: 3/3/2010

March 2010

General Studies Course Proposal Documentation

GPH 471 – Geographics: : Interactive and Animated Cartography and Geovisualization

Table of Contents:

Course Description

This course will introduce the concepts of geovisualization, animated and interactive cartography in a computationally based manner. To support the latter a major component of the first half of the course will be a primer on Python for geovisualization and gecomputation. The second half of the course takes the form of a studio where students will be challenged to apply their newly aquired technical skill sets to individualized projects in exploratory geovisualization.

Proposer: Please complete the following section and attach appropriate documentation.

ASU--[CS] CRITERIA			
A COMPUTER/STATISTICS/QUANTITATIVE APPLICATIONS [CS] COURSE MUST SATISFY ONE OF THE FOLLOWING CRITERIA: 1, 2, OR 3			
YES	**NO**		**Identify Documentation Submitted**
		1. Computer applications*: courses must satisfy both **a** and **b**:	
☒	☐	**a.** Course involves the use of computer programming languages or software programs for quantitative analysis, modeling, simulation, animation, or statistics.	The syllabus and course schedule have been annotated in yellow to indicate where this criteria has been met
		b. Course requires students to analyze and implement procedures that are applicable to at least one of the following problem domains **(check those applicable)**:	
☐	☐	**i.** Spreadsheet analysis, systems analysis and design, and decision support systems.	
☒	☐	**ii.** Graphic/artistic design using computers.	Course project is attached. This exemplifies how this criteria is met.
☐	☐	**iii.** Music design using computer software.	
☒	☐	**iv.** Modeling, making extensive use of computer simulation.	Exercise 1 is attached which exemplifies how this criteria is met.
☒	☐	**v.** Statistics studies stressing the use of computer software.	Exercise 2 is attached which exemplifies how this criteria is met.
*The **computer applications** requirement **cannot** be satisfied by a course, the content of which is restricted primarily to word processing or report preparation skills; learning a computer language or a computer software package; or the study of the social impact of computers. Courses that emphasize the use of a computer software package or the learning of a computer programming language are acceptable, provided that students are required to understand, at an appropriate level, **the theoretical principles embodied in the operation of the software and are required to construct, test, and implement procedures that use the software to accomplish tasks in the applicable problem domains.**			
		2. Statistical applications: courses must satisfy both **a** and **b**.	
☐	☐	**a.** Course has a minimum mathematical prerequisite of College Mathematics, College Algebra, or Precalculus, or a course already approved as satisfying the MA requirement.	

Course Prefix	Number	Title	Designation
GPH	471	Geographics	**CS**

Organizer to explain how the course meets CS Criteria.

Criteria (from checksheet)	How course meets spirit (contextualize specific examples in next column)	Please provide detailed evidence of how course meets criteria (i.e., where in syllabus)
1a. Course involves the use of computer programming languages or software programs for quantitative analysis, modeling, simulation, animation, or statistics.	Designing and implementing effective maps and other geovisualizations requires students to be conversant with basic cartographic theory, principles of visualization, and their implementation in modern computer languages and platforms.	In **syllabus**, course objective states: *This course will introduce the concepts of geovisualization, animated and interactive cartography in a computationally based manner. To support the latter a major component of the first half of the course will be a primer on Python for geovisualization and gecomputation. The second half of the course takes the form of a studio where students will be challenged to apply their newly acquired technical skill sets to individualized projects in exploratory geovisualization*
1b ii. Course requires students to analyze and implement procedures that are applicable to … Graphic/artistic design using computers.	Among the course objectives are 1. Introduction to the application of geovisualization to different research domains 2. Acquire computational skills in using Python for rapid prototyping and scientific computing 3. Experience in designing, implementing and testing an empirical project that combines core theory with software development	Course Project: *Each student will carry out an individual project that applies the theory and methods introduced in the first part of the course.* *An important component of the course is your project that takes the theoretical and computational concepts introduced in the lectures and applies them in the development of a new, or enhanced, component in the ESTDA package STARS.*

-- Continued on next page --

1b Course requires students to analyze and implement procedures that are applicable to … **iv.** Modeling, making extensive use of computer simulation.	1 b iv. Issues of representation and underlying data structures are germane in the modeling of geospatial data.	1b iv. Exercise 1 is designed to familiarlize the students with the Python scripting language which is subsequently used to carry out additional exercises, illustrate theoretical concepts, and to develop course projects.
1b Course requires students to analyze and implement procedures that are applicable to … v. Statistical studies stressing the use of computer software	1 b v. Many areas of spatial analysis and geovisualization necessitate customized programming that extends existing off-the-shelf packages.	I b v. Exercise 2 is designed to have the students implement a fundamental operation in spatial analysis, finding the shortest paths between nodes on a network.

1. Course Introduction

Welcome to GPH 471 Geographics for the Fall 2009 Semester.

1.1. Coordinates

1.1.1. Instructor

Dr. Sergio J. Rey

1.1.1.1. Office Hours

My office is Coor 5612.

Office hours this semester will be

- Weds 9:30-10:30
- Thurs 9:30-10:30
- and by appointment

1.1.1.2. Contact

Phone: (619) 928-4499

email: srey@asu.edu

web: http://geoplan.asu.edu/rey

1.1.2. Class Meetings

Where: SCOB328

When: T Th 1:30-2:45 PM

1.1.3. Class Web Sites

Blackboard: https://myasucourses.asu.edu/

Notes (this page): http://toae.org/courses/gph471f09

1.2. Introduction

This course will introduce the concepts of geovisualization, animated and interactive cartography in a computationally based manner. To support the latter a major component of the first half of the course will be a primer on Python for geovisualization and gecomputation. The second half of the course takes the form of a studio where students will be challenged to apply their newly aquired technical skill sets to individualized projects in exploratory geovisualization.

1.3. Objectives of the course

1. Introduction to basic theory in cartography and geovisualization
2. Introduction to the application of geovisualization to different research domains
3. Acquire computational skills in using Python for rapid prototyping and scientific computing
4. Experience in designing, implementing and testing an empirical project that combines core theory with software development

1.3.1. Prerequisites (Minimum requirements)

- Interest in geovisualization
- Interest in general scientific programming
- A willingness to learn
- A grade of B or better in GPH 371 Introductory Cartography (or equivalent)

1.3.2. Text(s)

1.3.2.1. Required:

Slocum, T.A., R.B. McMaster, F.C. Kessler and H.H. Howard (2009) Thematic Cartography and Geovisualization. Prentice Hall, Upper Saddle River.

1.3.2.2. Recommended:

Vaingast, S. (2009) Beginning Python Visualization: Crafting Visual Transformation Scripts. Apress, Berkeley.

Other readings will be assigned during the semester and made available through the course Blackboard site.

1.4. Grading

Your course grade will be based on the following components.

1b iv		10/01	Numerical Programming				
1b iv	7	10/06	Numerical Programming				
1b ii		10/08	GUI Toolkits				
~~1b iv~~ 1a	8	10/13	Introduction to svn				
1b ii		10/15	GUI Toolkits II				
	9	10/20	**No Class EDA Meeting**		Exercise 4	Exercise 3	100
		10/22	Studio				
1b ii	10	10/27	Choropleth mapping (Yao)	Ch 14			
1b ii			Color and topography (Maliszewski)	Ch 10 11			
		10/29	Studio			Exercise 4	100
1b ii	11	11/03	Map Animation (Padegimas)	Ch 21 22			
1b ii			Cartograms (Liu)	Ch 19			
		11/05	Studio				
			Interim presentations				
1b ii	12	11/10	Web Mapping (Malizia)	Ch 24			
1b v			Uncertainty (Folch)	Ch 23			
		11/12	Studio				
1b v	13	11/17	Exploration-Multivariate (Wei)	Ch 18 22			
			Virtual Environments (Interlante)	Ch 25			
		11/19	Studio				
	14	11/24	Future Directions	Ch 26			
		11/26	Thanksgiving Break				
	15	12/01	Presentations			**Project**	500
		12/03	Presentations				
	16	12/08	Presentations				

Benjamin Cummings

Geography & Atmospheric Sciences

View larger cover

Thematic Cartography and Geovisualization, 3/E

Terry A. Slocum, University of Kansas
Robert B McMaster, University of Minnesota
Fritz C Kessler, Frostburg State University
Hugh H Howard, American River College

ISBN-10: 0132298341
ISBN-13: 9780132298346
Publisher: Prentice Hall
Copyright: 2009
Format: Cloth; 576 pp
Published: 04/04/2008
Status: Instock

2. Introduction to Python for Geovisualization

2.1. Why, What, Who, When

2.1.1. Why?

The goals of this component of the course are

1. Provide an introduction to Python
2. A primer for researchers on Python for Geovisualization

I am assuming the following

1. Some programming background
2. Willingness to explore

2.1.2. Why Open Source?

Guido van Rossum (Creator of Python)

> I see this as the essence of open source projects: The energy and creativity of many people with diverse goals together can work miracles!

Enhancing the Scientific Process

> Rey, S.J (2009) "Show me the code: Spatial analysis and open source." Journal of Geographical Systems (11) 191-2007.

2.1.3. What

Python Features:

- High-level
- Object-oriented
- Scalable
- Extensible
- Portable
- Easy to learn, read and maintain
- Robust
- Rapid prototyping tool
- Memory manager
- Interpreted and byte-compiled

- It’s Fun!

A common question is why Python instead of

- C, Fortran, C++
- Java, Perl, Ruby, Scheme, VB
- Matlab, Gauss, R, Mathmatica, Maple

One response is that Python is super glue

- no need to replace legacy code
- useful for heterogenous projects/data/languages

2.1.3.1. Testimonials

Guido van Rossum http://www.artima.com/intv/speed.html

> A 20,000-line Python program would probably be a 100,000-line Java or C++ program. It might be a 200,000-line C program, because C offers you even less structure. Looking for a bug or making a systematic change is much more work in a 100,000-line program than in a 20,000-line program. For smaller scales, it works in the same way. A 500-line program feels much different than a 10,000-line program.

Bruce Eckel http://www.artima.com/intv/tippingP.html

> One of my first real productive experiences with Python, beyond just playing around with the language, involved image processing. I wanted to resize some GIF files. Given my experience with other languages, I figured this task might take me half a day if I were lucky. Even if there were an existing image processing library in Python, I figured the library would be complicated and take significant time to understand. I discovered a Python library that did graphics manipulation, and to my surprise, resizing GIFs was as simple as you can imagine it could be. You create an object, call reformat, pass in some arguments, and you’re done. In C++, and even in Java, the ease of understanding a library is not really part of the culture. In Python it really is. Instead of taking a half a day, which was my best hope, after a half an hour, I couldn’t think of any more features to add to my program. And I was just stunned. I thought, oh, that’s what people mean when they talk about Python’s incredible productivity.

Steve Waterbury, Software Group Leader, NASA STEP Testbed

> NASA is using Python to implement a CAD/CAE/PDM repository and model management, integration, and transformation system which will be the core infrastructure for its next generation collaborative engineering environment. We chose Python because it provides maximum productivity, code that’s clear and easy to maintain, strong and extensive (and

growing!) libraries, and excellent capabilities for integration with other applications on any platform. All of these characteristics are essential for building efficient, flexible, scalable, and well-integrated systems, which is exactly what we need. Python has met or exceeded every requirement we've had

2.1.3.2. Personal Experience

- Glue
    - LaTeX + python = index for 500+ page book
    - LaTeX + python = subject-author index for IRSR
    - LaTeX + python + cgi = bibliometrics
- GeoComputation
    - Spatial econometrics: Monte Carlo papers
    - Exploration of new ideas -> papers
- Utility programming
    - Backup scripts
    - Automated grade reports
- Full-blown applications
    - Pjs: Python Journaling System
    - STARS: Space-Time Anaysis of Regional Systems
    - PySAL: Python Library for Spatial Analysis

2.1.4. Who and When

Origins

- Guido van Rossum 1989
- Origins in ABC
- Public distribution 1991
- Guido is a Monty Python fan

2.2. Installing

Python runs on many different platforms. You are encouraged to install Python on your own laptop or desktop to facilitate working outside of lab time. Although there are a number of places to get Python, we will be using a number of libraries that are not (yet) part of the official distribution of Python, and for which installation can be tedious. To simplify things you are encouraged to use the Enthought Python Distribution (EPD) The good folks at Enthought have done a huge amount of work gathering up many useful libraries and making this available in a single download.

2.2.1. Editor

You will benefit from becoming proficient in using a text editor to both create and debug your Python (and other) code. There are many to choose from but the two most popular seem to be:

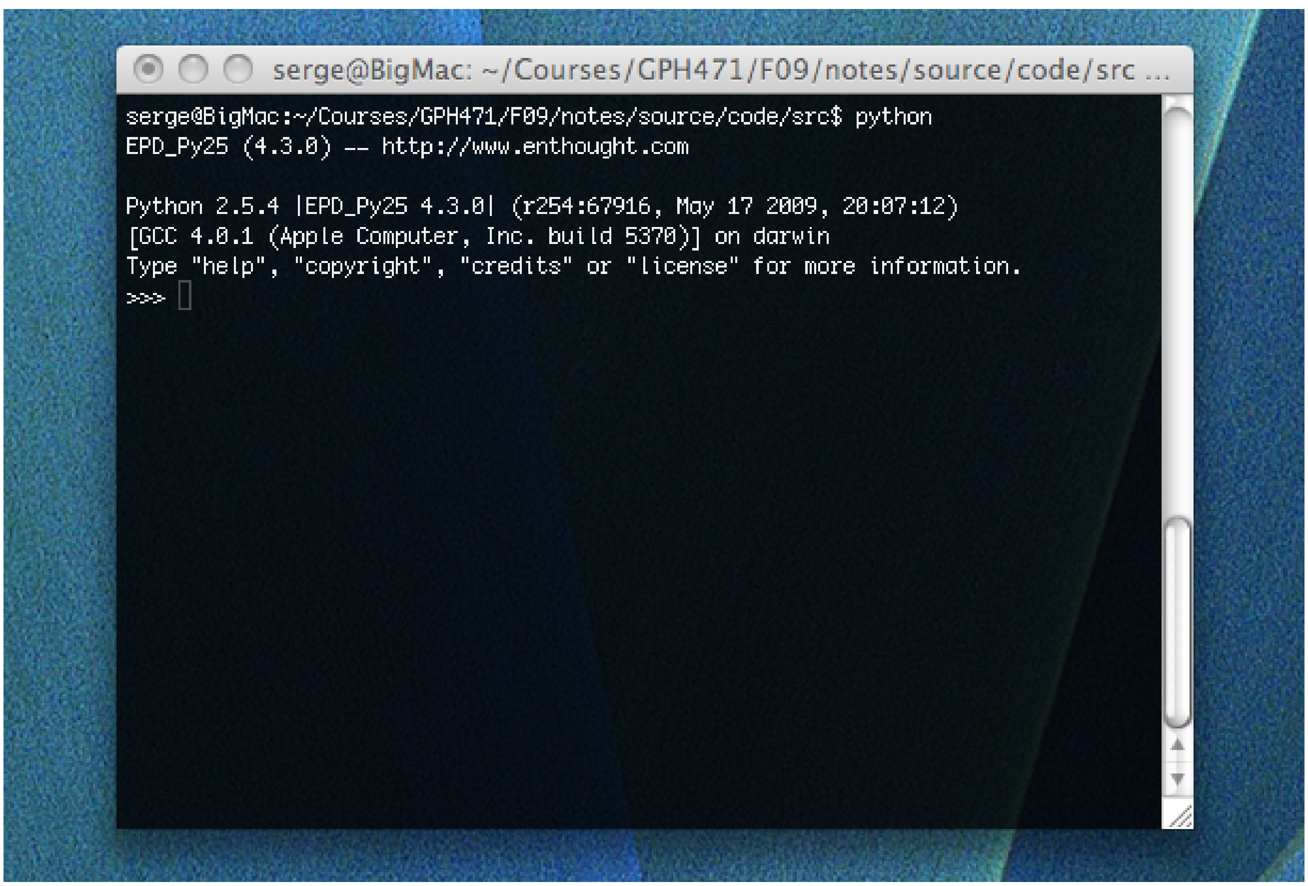
- Vim
- EMACS

You can also use IDLE which is the built in integrated development environment for Python. I use Vim so if you choose a different option for your editor, you are on your own ;->

2.3. Workflow, Python, and iPython

2.3.1. Python

I work with Python through a terminal and a text editor. An example of starting Python from a terminal follows:



To quit Python:

```
Python 2.5.4 |EPD_Py25 4.3.0| (r254:67916, May 17 2009, 20:07:12)
[GCC 4.0.1 (Apple Computer, Inc. build 5370)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>> exit()
serge@BigMac:~/Courses/GPH471/F09/notes/source/code/src$
```

and we are back at the prompt.

2.3.2. iPython

While you are free to use the regular Python interpretor, there is an enhanced interpreter available in iPython that we will use in the rest of the course. We start iPython in a similar fashion to Python:

```
serge@BigMac:~/Courses/GPH471/F09/notes/source/code/src$ ipython
Enthought Python Distribution -- http://code.enthought.com

Python 2.5.4 |EPD_Py25 4.3.0| (r254:67916, May 17 2009, 20:07:12)
Type "copyright", "credits" or "license" for more information.

IPython 0.9.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

2.3.3. Command line interaction

Issue a command:

```
In [1]: print 'Hello world'
Hello world

In [2]:
```

Creating an object and using introspection:

```
In [2]: s='Hello world'

In [3]: s

Out[3]: 'Hello world'

In [4]: s?
Type:           str
Base Class:     <type 'str'>
String Form:    Hello world
Namespace:      Interactive
```

```
Length:         11
Docstring:
    str(object) -> string

    Return a nice string representation of the object.
    If the argument is a string, the return value is the same object.


In [5]:
```

Everything in Python is an object with a value and id

```
In [5]: id(s)

Out[5]: 15703360

In [6]:
```

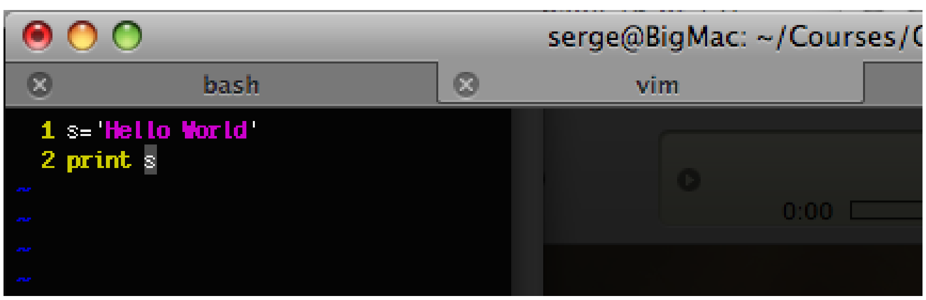
Recalling previous commands, use `ctrl-p`

```
In [5]: id(s)

Out[5]: 15703360

In [6]:

In [7]: id(s)
```

Put our commands in a file called `my_script.py`

```
In [9]: edit my_script.py
```

will bring up a vim session on my machine:



when I save that file and quit the editor (`command-mode :wq`) then iPython will try to run the script:

```
In [9]: edit my_script.py
Editing... done. Executing edited code...
Hello World
```

We can then either open up a separate window and keep the edited file there in an editor and switch back and forth between the editor and terminal running the iPython session, or call the editor directly from iPython as we did above. It all depends on whether you like to have two terminals open (one for editing and one for the interpretor, or only one terminal that you toggle between iPython and your editor). Lets make some changes to our file (be sure to save the changes):

```
s='Hello World'
print s

print id(s)
```

and then see what happens when we run this file:

```
In [11]: run my_script.py
Hello World
15771152
```

so our change is reflected.

2.3.3.1. Tab completion

One very handy feature of iPython is that it supports tab completion which can save us a great deal of typing. To see this, the command to run our script would be run my_script.py. Rather than typing that completely, first enter ru then a <tab>

```
In [12]: ru
%run     %runlog

In [12]: %run
```

so that gets us the first part, then enter a <space> followed by my then <tab> and then <return> to run our script:

```
In [12]: %run my_script.py
Hello World
9761520
```

2.3.3.2. Command History

We can get a record of everything we have done thus far using the history command:

```
[16]: hi
%hist     %history

In [16]: %hist
1 : print 'Hello world'
2 : s='Hello world'
3 : s
4 : #?s
```

```
5 : id(s)
6 :
7 : id(s)
8 :
9 : _ip.magic("edit my_script.py")
10: _ip.magic("edit my_script.py")
11: _ip.magic("run my_script.py")
12: _ip.magic("run my_script.py")
13:
14: #?%run
15: _ip.magic("runlog my_script.py")
16: _ip.magic("hist ")

In [17]:
```

2.3.3.3. Autologging

We can tell iPython to save our commands to a log file in case we want to revisit our interactive work later and use it as the building block for a larger script.

```
[17]: %logstart
Activating auto-logging. Current session state plus future input saved.
Filename       : ipython_log.py
Mode           : rotate
Output logging : False
Raw input log  : False
Timestamping   : False
State          : active

In [18]: s='a new string'

In [19]: len(s)

Out[19]: 12

In [20]: print s
a new string

In [21]:
Do you really want to exit ([y]/n)? y
```

This creates the file `ipython_log.py`:

```
serge@BigMac:~/Courses/GPH471/F09/notes$ ls *.py
ipython_log.py my_script.py
serge@BigMac:~/Courses/GPH471/F09/notes$ cat ipython_log.py
#log# Automatic Logger file. *** THIS MUST BE THE FIRST LINE ***
#log# DO NOT CHANGE THIS LINE OR THE TWO BELOW
#log# opts = Struct({'__allownew': True, 'logfile': 'ipython_log.py'})
#log# args = []
#log# It is safe to make manual edits below here.
#log#-----------------------------------------------------------------------
print 'Hello world'
s='Hello world'
s
#?s
```

```
id(s)

id(s)

_ip.magic("edit my_script.py")
_ip.magic("edit my_script.py")
_ip.magic("run my_script.py")
_ip.magic("run my_script.py")

#?%run
_ip.magic("runlog my_script.py")
_ip.magic("hist ")
_ip.magic("logstart ")

s='a new string'
len(s)
print s
```

We could then copy this file and edit to to reflect changes we one to add, and then run it as a Python script.

2.4. Getting Started Resources

This is enough to get you rolling. You are encouraged to check out other excellent tutorials to further your Python skills.

- The Python Tutorial [1]
- Fernando Perez's Starter Kit

2.5. Exercises

1. Install Python on your own personal desktop or laptop. On the Blackboard site, submit a pdf that includes:
   - a screen capture of the use of a terminal running iPython together with an editor editing a Python script
   - a copy of your ipython_log.py from a session where you are running your program and exploring the built-in documentation

Footnotes

[1] I know of several leading Python contributors who were attracted to the language by working through this tutorial. They were hooked after completing it, so you should work through it.

4. Conditional Execution and Functions

4.1. Control Flow

Controls the order in which code is executed.

4.1.1. if/else

```
>>> x=range(10)
>>> if len(x) > 5:
...     print 'length of x is greater than 5'
... else:
...     print 'length of x is less than or equal to 5'
...
length of x is greater than 5
>>>
```

A couple of things to note. First, blocks are delimited by indentation. Second, the first condition is true so the block following that condition is executed and the else statement is ignored. If the first condition were false, then the block following the else statement would have been executed:

```
>>> x=range(2)
>>> if len(x) > 5:
...     print 'length of x is greater than 5'
... else:
...     print 'length of x is less than or equal to 5'
...
length of x is less than or equal to 5
```

4.1.2. if/elif/else

We can implement a **switch** statement using:

```
>>> value=10
>>> if value < 5:
...     print 'less than five'
... elif value < 10:
...     print 'less than ten'
... else:
...     print 'greater than or equal to ten'
...
greater than or equal to ten
>>>
```

Again, once a condition is true, its code block is executed and we leave the conditional statement and continue on

4.1.3. for/range

These are used to iterate over sequences using an index:

```
>>> for i in range(4):
...     print i
...
0
1
2
3
>>>
```

or over values

```
>>> for name in ('waldo','rick','reg'):
...     print name
...
waldo
rick
reg
>>>
```

Sometimes we want to keep track of both the value and the index. This can be done with **enumerate**:

```
>>> for i,name in enumerate(('waldo','rick','reg')):
...     print i,name
...
0 waldo
1 rick
2 reg
>>>
```

We can also iterate over a dictionary:

```
>>> d={'weights':[1,2], 'neighbors':[98,101]}
>>> for key,value in d.items():
...     print key,value
...
neighbors [98, 101]
weights [1, 2]
>>>
```

4.1.4. while/break/condition

while can be used to execute a code block as long as a condition is true:

```
>>> names=['waldo','rick','reg']
>>> while names:
...     name=names.pop()
...     print name
```

```
...
reg
rick
waldo
>>>
```

Here the condition is that the list **names** is not empty.

break is used to jump out of the enclosing **while** loop:

```
>>> names=['waldo','rick','reg']
>>> i=0
>>> while names:
...     name=names.pop()
...     if i == 1:
...         break
...     print name
...     i+=1
...
reg
>>>
```

So here we are using **i** as a sentinel, keeping track of a second condition to evaluate to terminate the loop. As soon as that second condition is satisfied we jump out of the while loop and do no more printing.

We also introduced the increment operator **i+=1** which does the same thing as **i=i+1**

4.1.5. Conditional expressions

There are different types of conditional expressions we may use

4.1.5.1. if object

evaluates to **True** for any non-zero value for object, or if object is a sequence with length > 0

evalues to **False** for any zero value for object, or if object is a squence with length 0

4.1.5.2. a==b

Tests equality of two objects

```
>>> a=10
>>> b=10
>>> a==b
True
```

4.1.5.3. a in b

```
>>> b=range(10)
>>> a=11
>>> a in b
False
>>> a=5
>>> a in b
True
>>>
```

If b is a dictionary, **in** checks for the key

```
>>> b={'weights':[1,2], 'neighbors':[98,101]}
>>> 'neighbors' in b
True
>>>
```

4.2. Functional Programming

There are a number of very handy functional constructs in Python that bring great efficiencies in certain situations.

4.2.1. List Comprehensions

For loops on steroids

- very handy
- look a lot like for loops
- generate derived lists
- speed versus readability trade-off

```
>>> months="Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"
>>> years=range(2000,2010)
>>> calendar=["%s_%d"%(month,yr) for yr in years for month in months]
>>> len(calendar)
120
>>> print calendar
['Jan_2000', 'Feb_2000', 'Mar_2000', 'Apr_2000', 'May_2000', 'Jun_2000', 'Jul_2000',
>>>
```

4.2.2. List Filtering

We can also combine list comprehension with conditionals:

```
>>> x=range(30)
>>> odd_values = [ y for y in x if y%2 ]
>>> odd_values
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
>>>
```

4.2.3. map

map allows us to apply a function against each element of a sequence. This can be useful for simplifying code. Compare:

```
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> xs=[str(xi) for xi in x]
>>> xs
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

against a version using **map**:

```
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> xs=map(str,x)
>>> xs
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

4.2.4. zip

zip is a way to conjoin to sequences together. Think of two sides of a zipper and you get the idea:

```
>>> x=range(10)
>>> y=range(20,30)
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
>>> z=zip(x,y)
>>> z
[(0, 20), (1, 21), (2, 22), (3, 23), (4, 24), (5, 25), (6, 26), (7, 27), (8, 28), (9,
>>>
```

Note that the two sequences do not have to be the same length. If they are not, the resulting sequence will be truncated to the length of the shorter of the two origin sequences:

```
>>> y=range(20,25)
>>> z=zip(x,y)
>>> z
[(0, 20), (1, 21), (2, 22), (3, 23), (4, 24)]
>>> u=zip(y,x)
>>> u
[(20, 0), (21, 1), (22, 2), (23, 3), (24, 4)]
>>>
```

4.3. Functions

Functions are ways we can extend Python by writing code to add functionality that we would like to reuse. To do this we will put our functions in **modules**. Moreover, since functions, like everything, are objects in Python we can pass them around in flexible ways.

4.3.1. Components

- Name
- Argument(s)
- Return values

4.3.2. Defining functions

```
>>> def square(x):
...     return x*x
...
>>>
```

- `def`: Python keyword for function definition
- `square`: function name
- `x`: function argument
- `return`: what comes out of the function

4.3.3. Using functions

```
>>> square(8)
64
>>> x=square(10)
>>> x
100
>>>
```

4.3.4. Function Argument Types

- positional
- keyword
- variable length positional
- variable length keyword

4.3.4.1. Positional Arguments

```
>>> def power(x,exponent):
```

```
...     return x**exponent
```

In this function we have two arguments that are distinguished only by their position in the signature of the function. The position is critical:

```
>>> power(2,3)
8
>>> power(3,2)
9
>>>
```

In other words, the first argument passed in is assigned to the local variable `x`, while the second argument passed in is assigned to the local variable `exponent`.

Positional arguments are **required**. If we omit one we will get a traceback:

```
>>> power(3)
---------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
TypeError: power() takes exactly 2 arguments (1 given)
```

4.3.4.2. Keyword Arguments

Keyword parameters can serve two uses:

- define default values for parameters
- self document our functions

```
>>> def power(x=2,exponent=3):
...     return x**exponent
...
>>> power()
8
>>> power(x=4)
64
>>> power(exponent=4)
16
>>> power(x=8,exponent=2)
64
>>> power(exponent=2,x=8)
64
>>> power(8,2)
64
>>>
```

Note that in the last case we are using the keywords implicitly and their positions explicitly.

The other thing to keep in mind is that the default values for the keywords are evaluated when the function is defined, not when it is called:

```
>>> x=10
>>> power()
8
>>>
```

4.3.4.3. Passing by value

Arguments to functions are passed by value - in other words Python passes the object to which the variable refers, not the variable itself. If the value is immutable, the function does not modify the caller’s variable. If the value is mutable, the function modifies the caller’s variable:

```
>>> def bar(x,y):
...     x=23
...     y.append(17)
...     print 'x is ',x
...     print 'y is ',y
...
>>> a=10
>>> b=[40,-9]
>>> bar(a,b)
x is  23
y is  [40, -9, 17]
>>> a
10
>>> b
[40, -9, 17]
>>>
```

4.3.4.4. Positional and Keyword Arguments

You can combine these two types of arguments, using positional arguments to specify required parameters, while keyword arguments can be used to define optional parameters:

```
>>> def power(x,exponent=2):
...     return x**exponent
...
>>> power(2)
4
>>> power(7)
49
>>> power(2,exponent=3)
8
>>> power(2,3)
8
>>>
```

Note that the positional arguments have to precede the keyword arguments.

```
>>> power(exponent=3,2)
------------------------------------------------------------
```

```
   File "<ipython console>", line 1
SyntaxError: non-keyword arg after keyword arg (<ipython console>, line 1)
```

```
>>> power(exponent=2)
---------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
TypeError: power() takes at least 1 non-keyword argument (0 given)
```

4.3.4.5. Variable Length Positional Arguments

We might have reason to also include the ability for our functions to accept an undetermined (at definition time) number of positional arguments:

```
>>> def power(x,exponent=2,*names):
...     print 'x: ',x
...     print 'exponent: ',exponent
...     for name in names:
...         print name
...
>>> power(10)
x:  10
exponent:  2
>>> power(10,3)
x:  10
exponent:  3
>>> power(10,3,'alan','luc')
x:  10
exponent:  3
alan
luc
>>> power(10,3,'alan','luc',range(3))
x:  10
exponent:  3
alan
luc
[0, 1, 2]
>>>
```

The variable length positional parameters are tucked into a tuple.

4.3.4.6. Variable Length Keyword Arguments

```
>>> def power(x, **theRest):
...     print x
...     for key,value in theRest.items():
...         print key,value
...
>>> power(2)
2
>>> power(2,name='alan',title='professor')
2
name alan
title professor
>>>
```

The variable length positional parameters are tucked into a dictionary.

Positional arguments have to precede keyword arguments, which in turn have to precede variable length positional and variable length keyword arguments.

4.3.5. Functions as Objects

```
>>> def power(base,exponent=2):
...     return base**exponent
...
>>> def printMe(message):
...     print message
...
>>> x=4
>>> print power(x,3)
64
>>> f1=power
>>> print f1(x,3)
64
>>> functions={}
>>> functions['power']=power
>>> functions['printMe']=printMe
>>> functions['power'](2,3)
8
>>> functions['printMe']('a long string')
a long string
>>>
```

Tucking away functions in a dictionary and then using the dictionary to call the functions is known as **dispatching** and can be very handy.

4.4. Exercises

4.4.1. Shortest path

Write a function that finds the length of the shortest path between each pair of nodes on the following graph:

```
>>> g={0: [2, 1], 1: [0, 3], 2: [0, 4, 3], 3: [1, 2, 5], 4: [2, 6, 5], 5:
```

where the key is the id of the node, and the values of the key are the first order neighbors. In other words, node ‘0’ is connected to nodes ‘2’ and ‘1’, while node ‘4’ is connected to nodes ‘2’, ‘6’, and ‘5’. Assume all first order connections are of the same length (i.e., an unweighted graph). Your function needs to:

- clearly define the required arguments

- find the lengths of the shortest paths
- return the shortest paths lengths in a data structure of your choosing

4.4.2. Paths

Write a second function that returns the actual shortest paths along with their lengths.

Upload your code to the Blackboard site for grading.

3. Project Introduction

3.1. Introduction

An important component of the course is your project that takes the theoretical and computational concepts introduced in the lectures and applies them in the development of a new, or enhanced, component in the ESTDA package STARS.

What follows are some possible areas to choose from. These are not set in stone, but should give you an idea of the scope of topics that are possible. We may combine (or subdivide) some of these as the concepts are refined.

Final decisions about the specific requirements for each project and who is assigned which topic will be made on **October 15**. If you are interested in any of these (or have another possible topic) talk to me as soon as possible. The early bird gets the worm.

3.2. Components

Broadly speaking there are three general areas the components are organized into:

1. Algorithms and Analytical Modules
2. Data and Data Structures
3. Visualization

At first glance, you might think that only the third component is relevant to a course on geovisualization. However, the ability to support state of the science visualization methods rests fundamentally on the first two components and thus these are included here.

When considering which project to choose, keep in mind two things. First, the specific projects may overlap with others, so collaboration on those areas of overlap should be pursued to the benefit of both projects. Second, I will work closely with each of you on your specific module/projects.

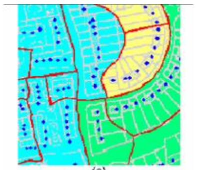
3.2.1. Algorithms and Analytical Modules

Work on the algorithms and analytical modules focuses on the computational aspects of visualization and ESTDA. If you are interested in algorithms, ESTDA or ESDA methods and related techniques, this is the area to work in. Although coding related to these components is not including visualization development in the form of code, you will be responsible for suggesting ways that the analytical components should be linked to visualization components (that others will be developing).

3.2.1.1. Clustering

Implement the following methods:

- agglomerative clustering
- partitive clustering
- kmeans
- Spatially constrained
    - AZP
    - max-p



(a)

Conceptualize different types of visualization methods associated with these methods. For example, dendrograms for agglomerative and partitive clustering, as well as silhouette plots. Focus on what type of interactions should be supported.

3.2.1.2. Classification

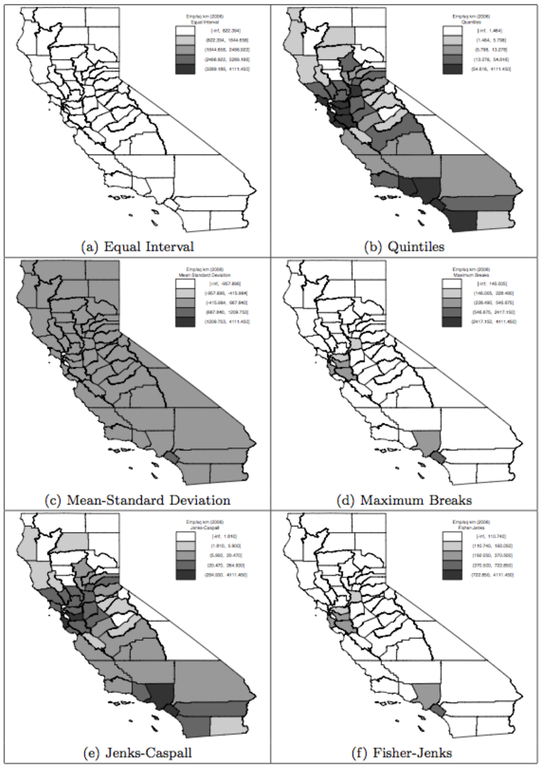


Figure 3.3: Alternative Map Classifications: California County Employment Density 2006

Tasks include:

- extending (and testing) existing classification module
- generalizing for use in other (beyond maps) modules
    - multivariate clustering
    - histograms

3.2.1.3. Markov

The Markov module needs an overhaul so that the existing methods are implemented in a cleaner fashion. Additionally, there are a number of new diagnostics, and tests that we can add to the modules to support new questions in ESTDA related to issues of asymmetry in the transitions, spatial poverty trap identification, testing Markov properties, working with thing cells, and others.

3.2.2. Data and Data Structures

3.2.2.1. Internal data structures for space-time data

Tasks include:

- research alternative data structures for space-time data (area units)
- implement two schemes of new data structures
- test two schemes
- make recommendation

The central issue is designing a system where these different types of interaction can coexist and providing an intuitive interface for the user to trigger the different types of interaction. For example, how does a user tell a view to start zooming versus panning, or selection for brushing versus selection for cumulative brushing.

Also the different types of interaction need to be broken down into graphical primitives. For example, highlighting a polygon, might be generalized or abstracted to a view method for setting the background color of a mark.

3.2.3.2. Signaling architecture

This is concerned with the communication between the different views to support the different types of interaction. It needs to support sending messages between views to signal what has happened on a source view so that destination views respond accordingly. This should be designed in an abstract way initially. Eventually there will be an integration with the project designing the view interaction system which will define the specific signals to be sent.

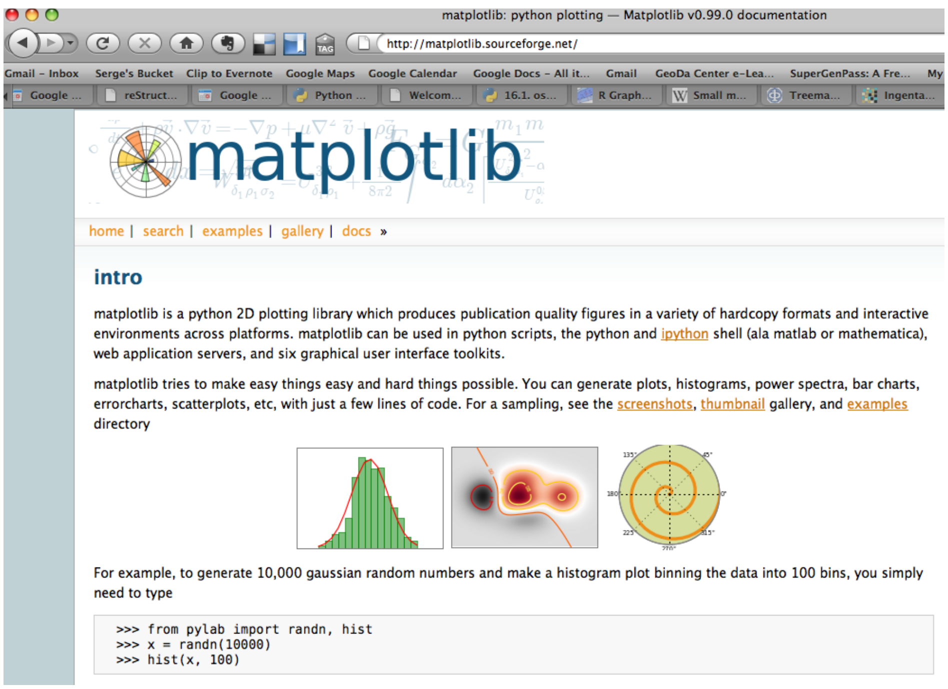
3.2.3.3. Comparison of Tkinter and WxPython

STARS is written using Tkinter as the GUI toolkit. This project will explore a comparison of using WxPython for the same purpose to provide the following:

- relative strengths of each toolkit for visualization
- relative weaknesses of each toolkit for visualization

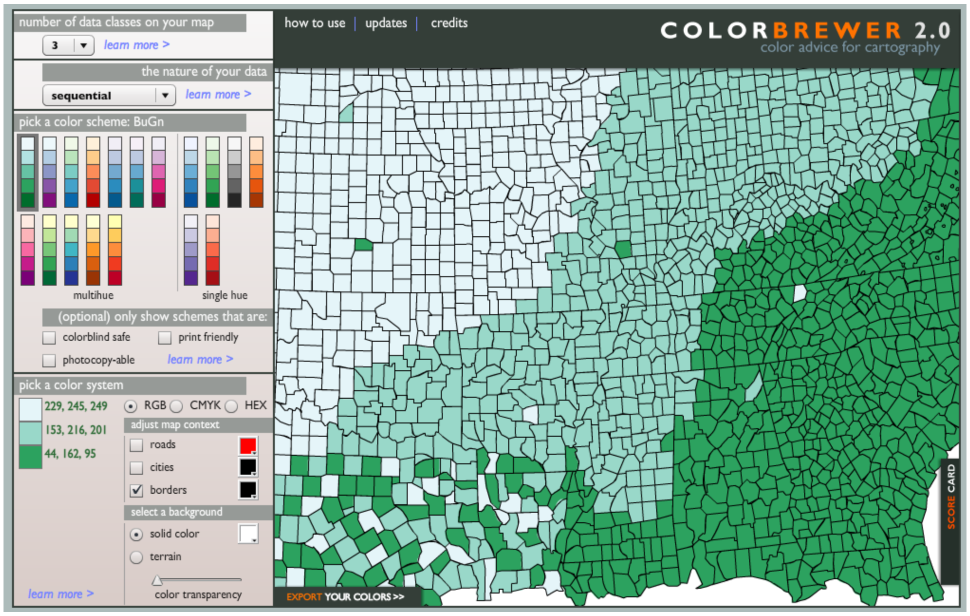
The comparison would be on a relative streamlined implementation of the visualization component of STARS, focusing on say a MAP and ad BOX-PLOT.

3.2.3.4. Evaluation of matplotlib



Matplotlib is a central component of scientific computing with Python. This project would evaluate Matplotlib for its suitability for implementation of the visualization layer for STARS

3.2.3.5. Color schemes and legends

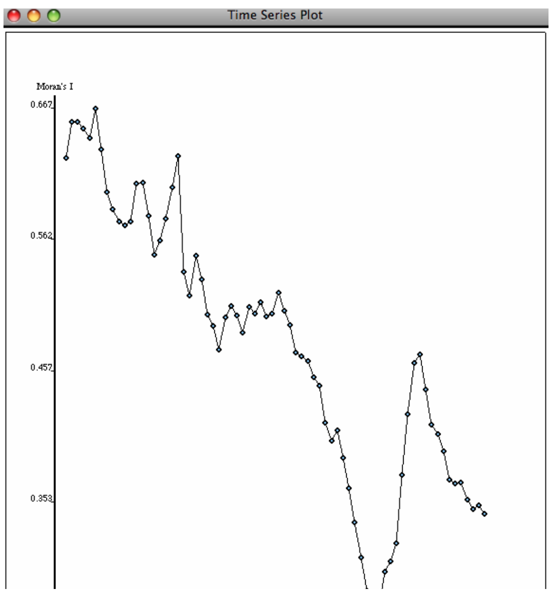


Tasks include

- new implementation of colorbrewer
- development of editable legends
- linking to map (and other) classifiers

3.2.3.6. Enhanced view: Multiple Time series

Currently STARS time series view only support a single series:

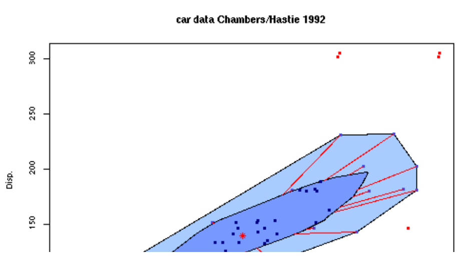


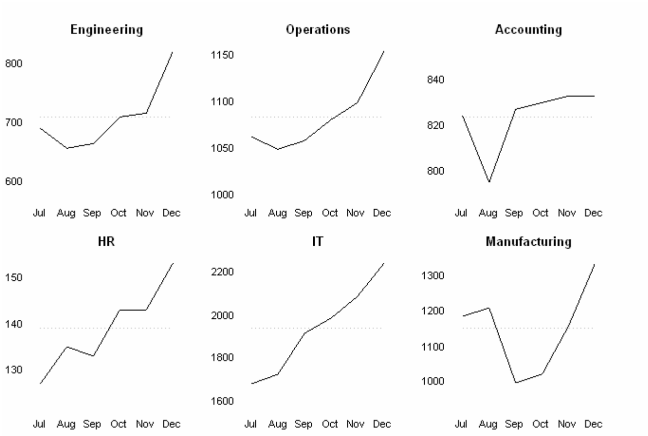
Extensions would include:

- ellipses for distribution of variable
- automated ordering of axes based on correlations
- user manipulation of axis
    - vertical orientation
    - horizontal orientation
    - flipping of axes
- application of color schemes

3.2.3.8. New view: Bagplot

The bagplot is a 2D generalization of the boxplot. The red asterisk is the bivariate median. The dark blue region is the bag, which contains 50% of the observations with greatest bivariate depth. The other blue region is called the loop, it contains observations that are in the fence (the bag expanded 3 times). Observations outside the fence are outliers regarding the concept of depth, they are too far away from the data's central bulk.



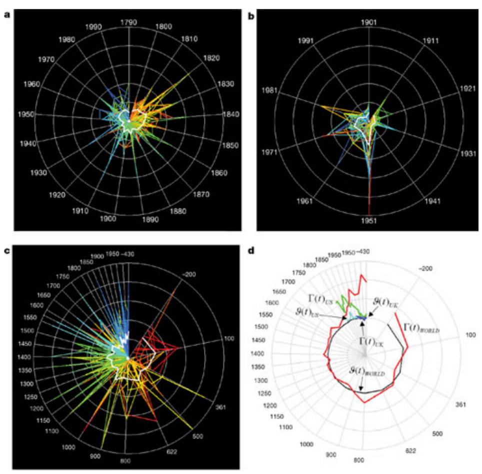


This project would be responsible for:

- determining what views in STARS are suitable for small-multiples
- designing an abstract architecture to support small-multiples
- implementation
- testing
- documentation

3.2.3.10. New view: Rank Clock

Rank clocks are essentially parallel coordinate plots, with the ordered axes representing time, that are then wrapped into a circle.

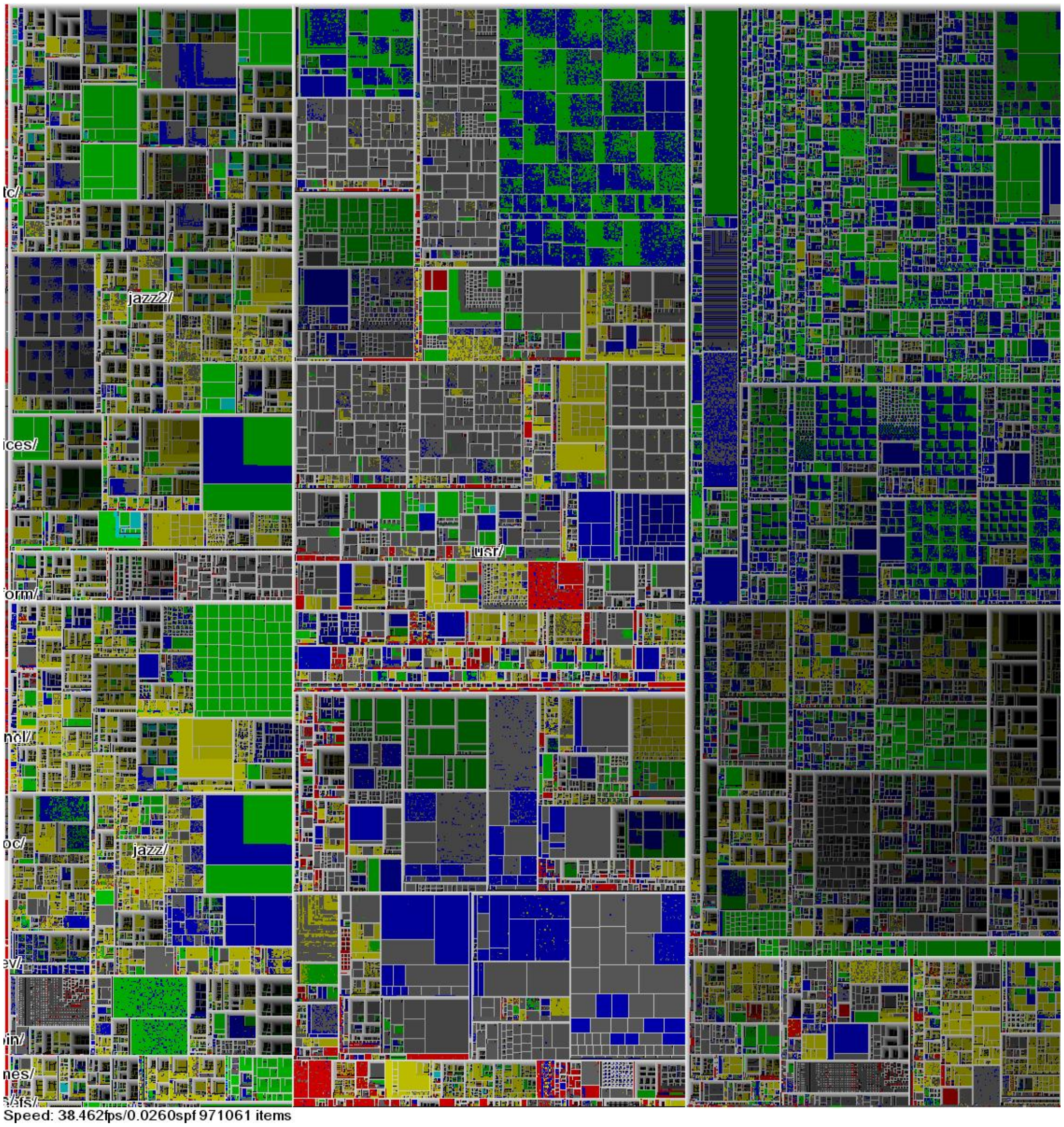


This project will:

- conceptualize how rank clocks may be used in STARS
- design the view, its interaction
- implement
- test
- document

3.2.3.11. New view: Lag-Lead graphs

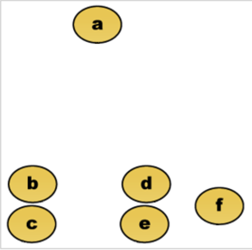
Main idea is to have two (or more) views be order in time such that when one moves forwards (or backwards) in time, the others do as well but the lag-lead relationship



This project will:

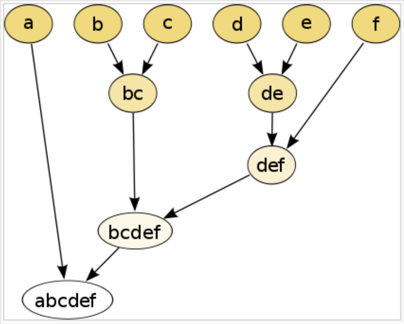
- conceptualize how treemaps graphs should be used in STARS
- design the view, its interaction
- implement
- test
- document

3.2.3.13. New view: Dendrogram



Raw data

The hierarchical clustering dendrogram would be as such:



Traditional representation

These are tree structures used to visualize the results of a hierarchical clustering algorithm. This project will design, implement and test a dendrogram view that:

- supports interactivity
- supports large n problems
    - nesting
    - collapsing
    - exploding
- supports changes in orientation

3.2.3.14. Drag and drop

Main idea: support the selection of say a polygon on a map and dragging that polygon (or sets of polygons) to a time series view to add the associated time series onto the second view.

Tasks include:

- conceptualizing different types of ESDA tasks this would facilitate
- designing interactions to support this conceptualization
- implementation
- testing
- documentation