quant-econ Documentation

Release 1

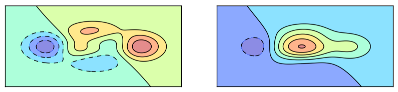
Thomas Sargent and John Stachurski

June 06, 2013

CONTENTS

This website presents a series of lectures on quantitative economic modeling using the Python programming language

The lectures are designed and written by John Stachurski and Thomas J. Sargent



CHAPTER

ONE

TABLE OF CONTENTS

1.1 Introduction to the Lectures

1.1.1 Overview

To be written – keep it brief

- What the lectures are about
- Why we wrote them
- What's different
- What you will learn

1.1.2 Structure

Brief explanation of the structure and contents of individual lectures

- Lecture x will teach you about Python
- Lecture y will show you how to get set up with Python
- Lecture z gives suggestions for those new to programming
- Lectures x–y provide a fast paced introduction to Python for those with some programming experience
- Etc.

Explain which lectures are more advanced, good ones to start with, etc.

1.2 Part 1: Programming in Python

This part of the course provides a relatively fast-paced introduction to the Python programming language

Our aim is not to teach you every facet of the Python language

Rather, we try to give you sufficent foundations for starting the applications, and then illustrate further features of the language using those applications as required

1.2.1 Lectures

About Python

Overview of This Lecture

In this lecture we will

- outline what Python is and what it can do
- showcase some of its abilities

It is **not** our intention that you

- seek to follow all the details
- try to replicate all you see

We will work through all of this material step by step in the lecture series

Our only objective for this lecture is to give you some feel of what Python can do

What is Python?

Python is a general purpose programming languange conceived in 1989 by Dutch computer programmer Guido van Rossum

Has experienced a huge increase in popularity in the last decade, and is now one of the most popular programming languages

Python is free and open source. All libraries of interest are completely free

Community-based development of the core language coordinated through the Python Software Foundation

Supported by a vast collection of standard and external software libraries

For reasons we will discuss, Python is particularly popular within the scientific community

Common Uses Python is a general purpose language used in almost all application domains

- communications
- web development
- CGI and graphical user interfaces
- games
- multimedia, data processing, security, etc., etc., etc.

Used extensively by Internet service and high tech companies such as

- YouTube
- Dropbox
- Industrial Light and Magic, etc., etc.

Often used to teach computer science and programming

- Introduction to computer science at MIT
- Computer science 101 at Udacity

Particularly popular within the scientific community (academia, NASA, CERN, etc.)

- Meteorology, computational biology, chemistry, machine learning, artificial intelligence, etc., etc.

Features

- A high level language suitable for rapid development
- Design philosophy emphasizes simplicity and readability
- Relatively small core language supported by many libraries (that can be imported at run time as required)
- A multiparadigm language, in that multiple programming styles are supported (object-oriented, functional, etc.)
- Interpreted rather than compiled

Scientific Programming

In recent years, Python has gained a huge amount of traction in the scientific community

This section briefly showcases some examples of Python for scientific programming

- All of these topics will be covered in detail later on
- Click on any figure to expand it

Numerical programming Core matrix and array processing capabilities are provided by the excellent NumPy library

NumPy provides the basic array data type plus some simple processing operations

For example

```
>>> import numpy as np                   # Load the library
>>> a = np.linspace(-np.pi, np.pi, 100)  # Create array (even grid from -pi to pi)
>>> b = np.cos(a)                        # Apply cosine to each element of a
>>> np.dot(a, b)                         # Compute inner product
-1.4460221214873 67e-14
```

The SciPy library is built on top of NumPy and provides additional functionality For example, let's calculate $\int_{-2}^{2} \phi(z) dz$ where ϕ is the standard normal density

```
>>> from scipy.stats import norm
>>> from scipy.integrate import quad
>>> phi = norm()
>>> value, error = quad(phi.pdf, -2, 2)  # Integrate using Gaussian quadrature
>>> value
0.9544997361036417
```

SciPy includes many of the standard routines used in

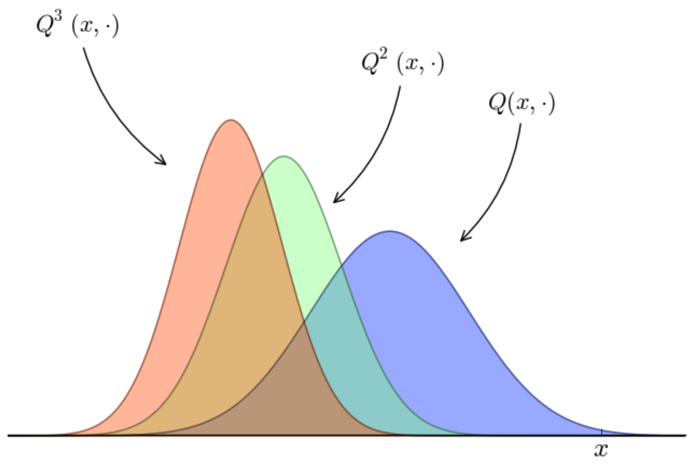
- linear algebra
- integration
- interpolation
- optimization
- distributions and random number generation
- signal processing

- etc., etc.

Graphics The most popular and comprehensive Python library for creating figures and graphs is Matplotlib

- Plots, histograms, contour images, 3D, bar charts, etc., etc.
- Output in many formats (PDF, PNG, EPS, etc.)
- LaTeX integration

Example 2D plot with embedded LaTeX annotations



Example contour plot

Example 3D plot

More examples can be found in the Matplotlib thumbnail gallery

Other graphics libraries include

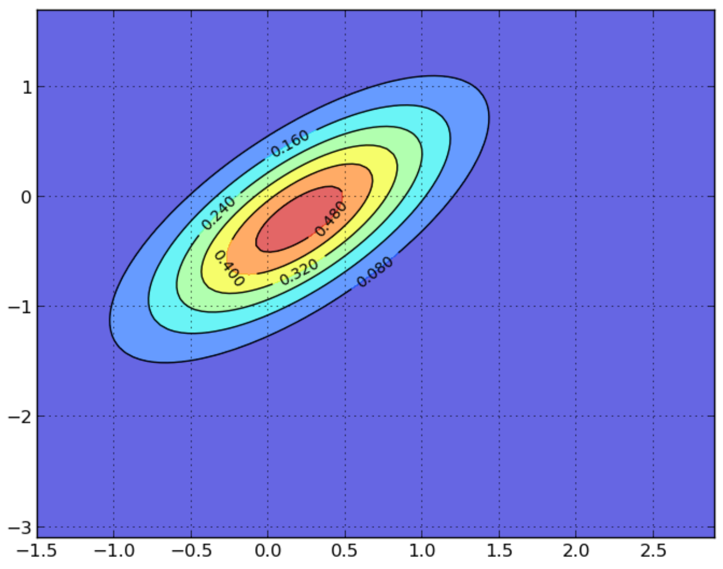
- VPython — 3D graphics and animations
- pyprocessing — a 'Processing'-like graphics environment
- Many more, but we will use only Matplotlib

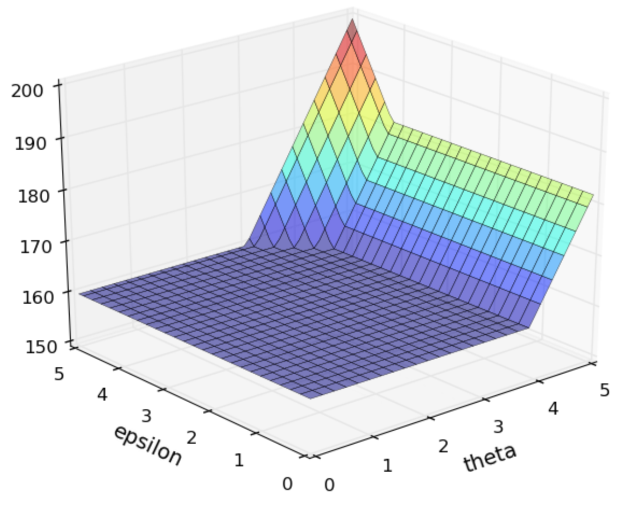
Symbolic Algebra Sometimes it's useful to be able to manipulate symbolic expressions in the spirit of Mathematica / Maple

The SymPy library provides this functionality from within the Python shell

```
>>> from sympy import Symbol
>>> x, y = Symbol('x'), Symbol('y')  # Treat 'x' and 'y' as algebraic symbols
>>> x + x + x + y
3*x + y
```

We can manipulate expressions


0.160
0.240
0.480
0.400
0.320
0.080
1
0
−1
−2
−3
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5


200
190
180
170
160
150
epsilon
theta
0
1
2
3
4
5

```
>>> expression = (x + y)**2
>>> expression.expand()
x**2 + 2*x*y + y**2
```

solve polynomials

```
>>> from sympy import solve
>>> solve(x**2 + x + 2)
[-1/2 + 7**(1/2)*I/2, -1/2 - 7**(1/2)*I/2]
```

and calculate limits, derivatives and integrals

```
>>> from sympy import limit, sin, diff
>>> limit(1 / x, x, 0)
oo
>>> limit(sin(x) / x, x, 0)
1
>>> diff(sin(x), x)
cos(x)
```

The beauty of importing this functionality into Python is that we are working within a fully fledged programming language

Can easily create tables of derivatives, generate LaTeX output, add it to figures, etc., etc.

Statistics Python also has excellent libraries for working with data

One of the most popular is Pandas

```
>>> import pandas as pd
>>> import scipy as sp
>>> data = sp.randn(5, 2)  # Create 5x2 matrix of random numbers for toy example
>>> dates = pd.date_range('28/12/2010', periods=5)
>>> df = pd.DataFrame(data, columns=('price', 'weight'), index=dates)
>>> print df
               price    weight
2010-12-28  0.007255  1.129998
2010-12-29 -0.120587 -1.374846
2010-12-30  1.089384  0.612785
2010-12-31  0.257478  0.102297
2011-01-01 -0.350447  1.254644
>>> df.mean()
price     0.176616
weight    0.344975
```

Pandas is fast, efficient, flexible and beautifully designed

Other useful statistics libraries:

- scikit-learn — machine learning in Python (sponsored by Google, among others)
- statsmodels — various statistical routines

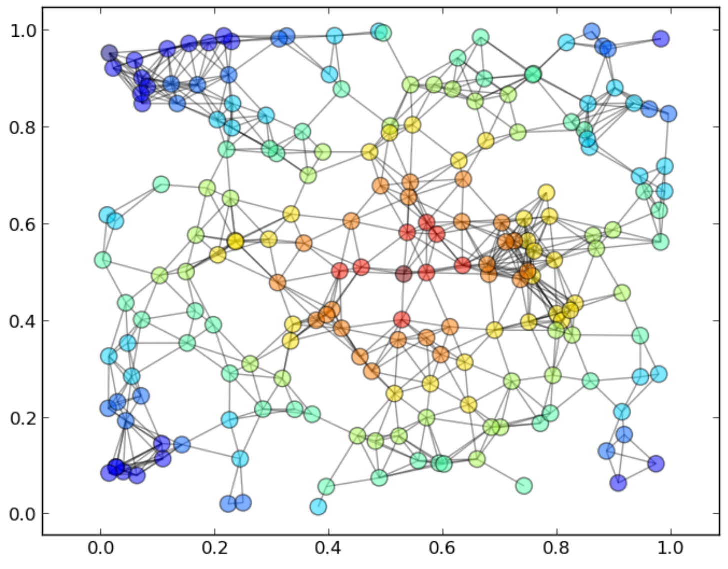
Networks and Graphs Python has libraries for studying graphs, perhaps the best known of which is NetworkX

- Standard graph algorithms for analyzing network structure, etc.
- Plotting routines
- etc., etc.

Here's some example code that generates and plots a random graph, with node color determined by shortest path length from a central node

```
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

G = nx.random_geometric_graph(200, 0.12)  # Generate random graph
pos = nx.get_node_attributes(G, 'pos')    # Get positions of nodes
# find node nearest the center point (0.5,0.5)
dists = [(x - 0.5)**2 + (y - 0.5)**2 for x, y in pos.values()]
ncenter = np.argmin(dists)
# Plot graph, coloring by path length from central node
p = nx.single_source_shortest_path_length(G, ncenter)
plt.figure()
nx.draw_networkx_edges(G, pos, alpha=0.4)
nx.draw_networkx_nodes(G, pos, nodelist=p.keys(),
                       node_size=120, alpha=0.5,
                       node_color=p.values(), cmap=plt.cm.jet_r)
plt.show()
```

The figure it produces looks as follows



Cloud Computing Running your Python code on massive servers in the cloud is becoming easier and easier

An excellent example is PiCloud

- Designed primarily for high performance scientific computing, including parallel processing
- Environment contains the scientific libraries described above (NumPy, SciPy, Pandas, etc.)
- Simple interface
- Built on top of Amazon Web Services

See also

- Pythonanywhere
- Wakari
- Amazon Elastic Compute Cloud
- The Google App Engine (Python, Java or Go)

Parallel Processing Apart from the cloud computing options listed above, you might like to consider

- Parallel computing through IPython clusters
- The Starcluster interface to Amazon's EC2
- GPU programming through Copperhead or Pycuda

Interfacing with C or Fortran But isn't Fortran / C faster for scientific computing (along with everything else)?

In one sense the answer is "yes"—these languages are much closer to the hardware, giving more control

But remember that *your* time is a far more valuable resource than the computer's time

The correct objective function to minimize is `total time = writing and debugging time + run time`

An ideal language would minimize both terms on RHS, but there is a *trade-off* here

- To minimize the first term, optimize for humans (a nice discussion can be found here)
- To minimize the second term, optimize for computers

Higher level languages such as Python are optimized for humans

Lower level languages such as Fortran and C are optimized for computers

Lower level languages run faster and give greater control, at the cost of

- taking longer to learn, write and debug
- more details to address (declaring variables, memory allocation/deallocation, etc.)
- requiring boilerplate code, writing of which is error prone and tedious

For these reasons, the modern scientific paradigm is to combine the strengths of high and low level languages as follows:

1. Write a prototype program in a high-level language such as Python
2. *If* the program is too slow, then profile it to find out where the bottlenecks are
3. Rewrite those *and only those* small parts of the code in Fortran / C
4. Rewrite the existing Python program to call this new Fortran / C code when necessary

There are many ways to accomplish this — we will cover several in a later lecture

Other Developments There are many other interesting developments with scientific programming in Python

Some representative examples include

- Blaze — a generalization of NumPy
- IPython notebook — Python in your browser with code cells, embedded images, etc.

- PyMC — Bayesian statistical modeling in Python
- Numba — speed up scientific Python code
- PyTables — manage large data sets
- CVXPY — convex optimization in Python

Setting up Your Python Environment

The objective of this lecture is to

1. Get a Python environment up and running with all the necessary tools
2. Install the Python programs that underpin these lectures

Important: Please read the following carefully

- The standard package is easy to install, but probably not what you want
- We need the scientific programming stack, which the core installation doesn't provide
- There are several different ways to set up a Python environment with these scientific tools
- Finding the right set up will make a huge difference to your productivity and enjoyment

Even if you have a working set up with scientific tools, it's well worth looking at the other possibilities

This lecture will work you through the different options

Quick Overview

Before we start it's useful to look at the big picture

- What does the Python environment consist of?
- How can we interact with the Python environment?

Notes: Provide simple instructions on how to

- unpack in a suitable directory
- set the Python pwd / path to that directory

The Python Environment To be written

Interacting with Python To be written

The IPython Notebook To be written

Set Option 1: Local installation

To be written

Set Option 2: Python in the Cloud

To be written

An Introductory Example

We're now ready to start learning the Python language itself, and the next few lectures are devoted to this task

Our approach is aimed at those who already have **at least some** knowledge of fundamental programming concepts, such as

- variables
- for loops, while loops
- conditionals (if/else)

Don't give up if you have no programming experience—you are not excluded

You just need to cover some of the fundamentals of programming before returning here

Two good references for first time programmers are

- Learn Python the Hard Way
- How to Think Like a Computer Scientist — the first 5 or 6 chapters

Overview of This Lecture

In this lecture we will write and then pick apart small Python programs

The objective is to introduce you to basic Python syntax and data structures

Deeper concepts—how things work—will be covered in later lectures

In reading the following, you should be conscious of the fact that all "first programs" are to some extent contrived

We try to avoid this, but nonetheless

- Be aware that we have written the programs to illustrate certain concepts
- By the time you finish the course, you will be writing the same programs in a rather different—and more efficient—way

What you should know before you start

- How to get a copy of the programs written for this course
- How to run these (or any other) Python programs through the Python interpreter

If you're not sure about one or the other, then please return to *this lecture*

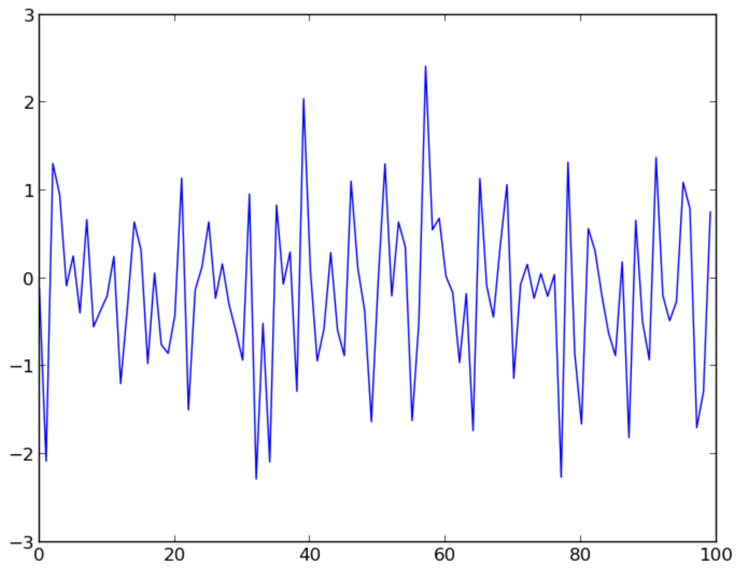
First Example: Plotting a White Noise Process

To begin, let's suppose that we want to simulate and plot the white noise process $\epsilon_0, \epsilon_1, \ldots, \epsilon_T$, where each draw ϵ_t is independent standard normal

In other words, we want to generate figures that look something like this:

Here's a program that accomplishes what we want

```python
import pylab
from random import normalvariate
ts_length = 100
epsilon_values = []   # An empty list
for i in range(ts_length):
    e = normalvariate(0, 1)
```



```
7        epsilon_values.append(e)
8    pylab.plot(epsilon_values, 'b-')
9    pylab.show()
```

The program can be found in the file `test_program_1.py` from the main repository

In brief,

- Lines 1–2 use the Python `import` keyword to pull in functionality from external libraries
- Line 3 sets the desired length of the time series
- Line 4 creates an empty list called `epsilon_values`, which will store the ϵ_t values as we generate them
- Line 5 tells the Python interpreter that it should cycle through the block of indented lines (lines 6–7) `ts_length` times before continuing to line 8
  - Lines 6–7 draw a new value ϵ_t and append it to the end of the list `epsilon_values`
- Lines 8–9 generate the plot and display it to the user

Let's now break this down and see how the different parts work

Import Statements First, consider the lines

```
1    import pylab
2    from random import normalvariate
```

Here `pylab` and `random` are two separate "modules"

- A *module* is a file, or a hierachy of linked files, containing code that can be read by the Python interpreter
- Importing a module causes the Python interpreter to run the code in those files

After importing a module, we can access anything defined within the module via `module_name.attribute_name` syntax

```
>>> import random
>>> random.normalvariate(0, 1)
-0.12451500570438317
>>> random.uniform(-1, 1)
0.35121616197003336
```

Alternatively, we can import attributes from the module directly

```
>>> from random import normalvariate, uniform
>>> normalvariate(0, 1)
-0.38430990243287594
>>> uniform(-1, 1)
0.5492316853602877
```

Both approaches are in common use

Lists Next let's consider the statement `epsilon_values = []`, which creates an empty list

Lists are a native Python data structure used to group a collection of objects. For example

```
>>> x = [10, 'foo', False]  # We can include heterogeneous data inside a list
>>> type(x)
<type 'list'>
```

Here the first element of `x` is an integer, the next is a string and the third is a Boolean value

When adding a value to a list, we can use the syntax `list_name.append(some_value)`

```
>>> x
[10, 'foo', False]
>>> x.append(2.5)
>>> x
[10, 'foo', False, 2.5]
```

Here `append()` is what's called a *method*, which is a function "attached to" an object—in this case, the list `x`

We'll learn all about methods later on, but just to give you some idea,

- Python objects such as lists, strings, etc. all have methods that are used to manipulate the data contained in the object
- String objects, have string methods, list objects have list methods, etc.

Another useful list method is `pop()`

```
>>> x
[10, 'foo', False, 2.5]
>>> x.pop()
2.5
>>> x
[10, 'foo', False]
```

The full set of list methods can be found here

Following C, C++, Java, etc., lists in Python are zero based

```
>>> x
[10, 'foo', False]
>>> x[0]
10
>>> x[1]
'foo'
```

Returning to `test_program_1.py` above, we actually create a second list besides `epsilon_values`

In particular, line 5 calls the `range()` function, which creates sequential lists of integers

```
>>> range(4)
[0, 1, 2, 3]
>>> range(5)
[0, 1, 2, 3, 4]
```

The For Loop Now let's consider the `for` loop in `test_program_1.py`, which we repeat here for convenience, along with the line that follows it

```
for i in range(ts_length):
    e = normalvariate(0, 1)
    epsilon_values.append(e)
pylab.plot(epsilon_values, 'b-')
```

The `for` loop causes Python to execute the two indented lines a total of `ts_length` times before moving on

These two lines are called a `code block`, since they comprise the "block" of code that we are looping over

Unlike most other languages, Python knows the extent of the code block *only from indentation*

In particular, the fact that indentation decreases after line `epsilon_values.append(e)` tells Python that this line marks the lower limit of the code block

More on indentation below—for now let's look at another example of a `for` loop

```
animals = ['dog', 'cat', 'bird']
for animal in animals:
    print("The plural of " + animal + " is " + animal + "s")
```

If you put this in a text file and run it you will see

```
The plural of dog is dogs
The plural of cat is cats
The plural of bird is birds
```

This example helps to clarify how the `for` loop works: When we execute a loop of the form

```
for variable_name in sequence:
    <code block>
```

The Python interpreter performs the following:

- For each element of `sequence`, it "binds" the name `variable_name` to that element and then executes the code block

The `sequence` object can in fact be a very general object, as we'll see soon enough

Code Blocks and Indentation In discussing the `for` loop, we explained that the code blocks being looped over are delimited by indentation

In fact, in Python **all** code blocks (i.e., those occuring inside loops, if clauses, function definitions, etc.) are delimited by indentation

Thus, unlike most other languages, whitespace in Python code affects the output of the program

Once you get used to it, this is a very good thing because it

- forces clean, consistent indentation, which improves readability

- removes clutter, such as the brackets or end statements used in other languages

On the other hand, it takes a bit of care to get right, so please remember:

- The line before the start of a code block always ends in a colon
  - `for i in range(10):`
  - `if x > y:`
  - `while x < 100:`
  - etc., etc.
- All lines in a code block **must have the same amount of indentation**
- The Python standard is 4 spaces, and that's what you should use

One small "gotcha" here is the mixing of tabs and spaces

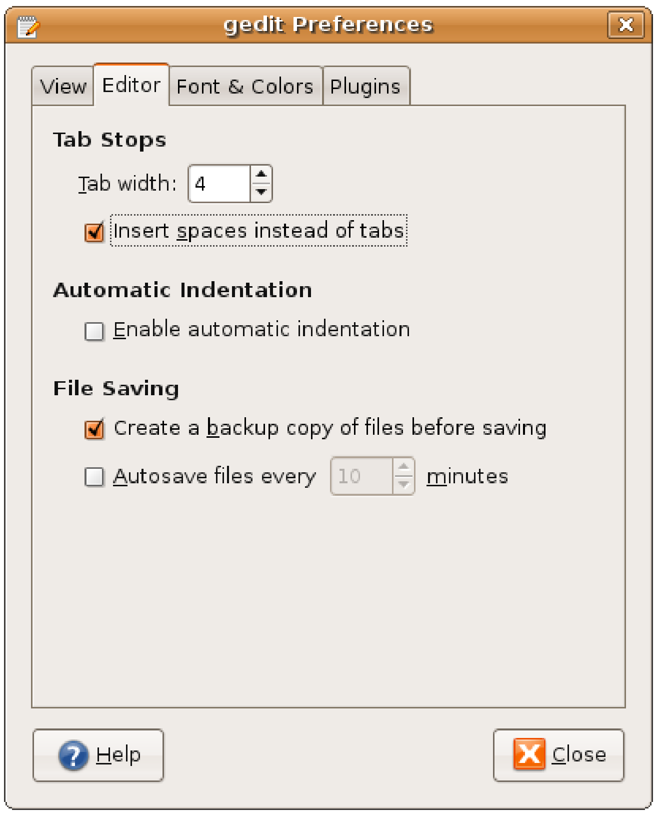
(Important: tabs and spaces are different entities in text files.)

You can use your `Tab` key to insert 4 spaces, but you need to make sure it's configured to do so

If you use a Python-specific editor such as Spyder or IPython Notebook then this will be taken care of for you

If you use a generic text editor then you might need to do this manually—a Google search will probably tell you how

Here's a screenshot of correct tab configuration for the Gedit text editor



Tab Stops

Tab width: 4

[✓] Insert spaces instead of tabs

Automatic Indentation

[ ] Enable automatic indentation

File Saving

[✓] Create a backup copy of files before saving

[ ] Autosave files every 10 minutes

While Loops The `for` loop is the most common technique for iteration in Python

But, for the purpose of illustration, let's modify `test_program_1.py` to use a `while` loop instead

In Python, the `while` loop syntax is as shown in the file `test_program_2.py` below

```
1  import pylab
2  from random import normalvariate
3  ts_length = 100
4  epsilon_values = []
5  i = 0
6  while i < ts_length:
7      e = normalvariate(0, 1)
8      epsilon_values.append(e)
9      i = i + 1
10 pylab.plot(epsilon_values, 'b-')
11 pylab.show()
```

The output of `test_program_2.py` is identical to `test_program_1.py` above (modulo randomness)

Comments:

- The code block for the `while` loop is lines 7–9, again delimited only by indentation
- The statement `i = i + 1` can be replaced by `i += 1`

User-Defined Functions Now let's go back to the `for` loop, but restructure our program to make the logic clearer

To this end, we will break our program into two parts:

1. A *user-defined function* that generates a list of random variables
2. **The main part of the program, which**
    (a) calls this function to get data
    (b) plots the data

This is accomplished in `test_program_3.py`

```
1  import pylab
2  from random import normalvariate
3
4  def generate_data(n):
5      epsilon_values = []
6      for i in range(n):
7          e = normalvariate(0, 1)
8          epsilon_values.append(e)
9      return epsilon_values
10
11 data = generate_data(100)
12 pylab.plot(data, 'b-')
13 pylab.show()
```

Let's go over this carefully, in case you're not familiar with functions and how they work

We have defined a function called `generate_data()`, where the definition spans lines 4–9

- `def` on line 4 is a Python keyword used to start function definitions
- `def generate_data(n):` indicates that the function is called `generate_data`, and that it has a single argument `n`
- Lines 5–9 are a code block called the *function body*—in this case it creates an iid list of random draws using the same logic as before
- Line 9 indicates that the list `epsilon_values` is the object that should be returned to the calling code

This whole function definition is read by the Python interpreter and stored in memory

When the interpreter gets to the expression `generate_data(ts_length)` in line 12, it executes the function body (lines 5–9) with `n` set equal to the value of `ts_length` (in this case, `100`)

The net result is that the name `data` on the left-hand side of line 12 is set equal to the list `epsilon_values` returned by the function

Conditions Our function `generate_data()` is rather limited

Let's make it slightly more useful by giving it the ability to return either standard normals or uniform random variables on $(0, 1)$ as required

This is achieved in `test_program_4.py` by adding the argument `generator_type` to `generate_data()`

```python
import pylab
from random import normalvariate, uniform

def generate_data(n, generator_type):
    epsilon_values = []
    for i in range(n):
        if generator_type == 'U':
            e = uniform(0, 1)
        else:
            e = normalvariate(0, 1)
        epsilon_values.append(e)
    return epsilon_values

data = generate_data(100, 'U')
pylab.plot(data, 'b-')
pylab.show()
```

Comments:

- Hopefully the syntax of the if/else clause is self-explanatory, with indentation again delimiting the extent of the code blocks
- We are passing the argument `U` as a string, which is why we write it as `'U'`
- Notice that equality is tested with the == syntax, not =
  - For example, the statement `a = 10` assigns the name `a` to the value `10`
  - The expression `a == 10` evaluates to either `True` or `False`, depending on the value of `a`

Now, there are two ways that we can simplify `test_program_4`

First, Python accepts the following conditional assignment syntax

```python
>>> x = -10
>>> s = 'negative' if x < 0 else 'nonnegative'
>>> s
'negative'
```

which leads us to `test_program_5.py`

```python
import pylab
from random import normalvariate, uniform

def generate_data(n, generator_type):
    epsilon_values = []
    for i in range(n):
```

```
7           e = uniform(0, 1) if generator_type == 'U' else normalvariate(0, 1)
8           epsilon_values.append(e)
9       return epsilon_values
10
11  data = generate_data(100, 'U')
12  pylab.plot(data, 'b-')
13  pylab.show()
```

Second, and more importantly, we can get rid of the conditionals all together by just passing the desired generator type *as a function*

To understand this, consider `test_program_6.py`

```
1   import pylab
2   from random import normalvariate, uniform
3
4   def generate_data(n, generator_type):
5       epsilon_values = []
6       for i in range(n):
7           e = generator_type(0, 1)
8           epsilon_values.append(e)
9       return epsilon_values
10
11  data = generate_data(100, uniform)
12  pylab.plot(data, 'b-')
13  pylab.show()
```

The only lines that have changed here are lines 7 and 11

In line 11, when we call the function `generate_data()`, we pass `uniform` as the second argument

The object `uniform` is in fact a **function**, defined in the `random` module

```
>>> from random import uniform
>>> uniform(0, 1)
0.2981045489306786
```

When the function call `generate_data(ts_length, uniform)` on line 11 is executed, Python runs the code block on lines 5–9 with `n` equal to `ts_length` and the name `generator_type` "bound" to the function `uniform`

- While these lines are executed, the names `generator_type` and `uniform` are "synonyms", and can be used in identical ways

This principle works more generally—for example, consider the following piece of code

```
>>> max(7, 2, 4)    # max() is a built-in Python function
7
>>> m = max
>>> m(7, 2, 4)
7
```

Here we created another name for the built-in function `max()`, which could then be used in identical ways

In the context of our program, the ability to bind new names to functions means that there is no problem *passing a function as an argument to another function*—as we do in line 12

List Comprehensions Now is probably a good time to tell you that we can simplify the code for generating the list of random draws considerably by using something called a *list comprehension*

List comprehensions are an elegant Python tool for creating lists

Consider the following example, where the list comprehension is on the right-hand side of the second line

```
>>> animals = ['dog', 'cat', 'bird']
>>> plurals = [animal + 's' for animal in animals]
>>> plurals
['dogs', 'cats', 'birds']
```

Here's another example

```
>>> range(8)
[0, 1, 2, 3, 4, 5, 6, 7]
>>> doubles = [2 * x for x in range(8)]
>>> doubles
[0, 2, 4, 6, 8, 10, 12, 14]
```

With the list comprehension syntax, we can simplify the lines

```
epsilon_values = []
for i in range(n):
    e = generator_type(0, 1)
    epsilon_values.append(e)
```

into

```
epsilon_values = [generator_type(0, 1) for i in range(n)]
```

Using the Scientific Libraries As discussed at the start of the lecture, our example is somewhat contrived

In practice we would use the scientific libraries, which can generate large arrays of independent random draws much more efficiently

For example, try

```
>>> from numpy.random import randn
>>> epsilon_values = randn(5)
>>> epsilon_values
array([-0.15591709, -1.42157676, -0.67383208, -0.45932047, -0.17041278])
```

We'll discuss these scientific libraries a bit later on

Exercises

Exercise 1 Recall that $n!$ is read as "n factorial" and defined as $n! = n \times (n-1) \times \cdots \times 2 \times 1$

There are functions to compute this in various modules, but let's write our own version as an exercise

In particular, write a function `factorial` such that `factorial(n)` returns $n!$ for any integer n

Solution: *View solution*

Exercise 2 The binomial random variable $Y \sim Bin(n, p)$ represents the number of successes in n binary trials, where each trial succeeds with probability p

Without any import besides `from random import uniform`, write a function `binomial_rv` such that `binomial_rv(n, p)` generates one draw of Y

Hint: If U is uniform on $(0, 1)$ and $p \in (0, 1)$, then the expression `U < p` evaluates to `True` with probability p

Solution: *View solution*

Exercise 3 Compute an approximation to π using Monte Carlo. Use no imports besides

```
from random import uniform
from math import sqrt
```

Your hints are as follows:

- If U is a bivariate uniform random variable on the unit square $(0,1)^2$, then the probability that U lies in a subset B of $(0,1)^2$ is equal to the area of B
- If $U_1, \ldots, U_n$ are iid copies of U, then, as n gets large, the fraction that fall in B converges to the probability of landing in B
- For a circle, area = pi * radius^2

Solution: *View solution*

Exercise 4 Write a program which prints one random outcome of the following game:

- Flip an unbiased coin 10 times
- If 3 consecutive heads occur one or more times within this sequence, pay one dollar
- If not, pay nothing

Use no import besides `from random import uniform`

Solution: *View solution*

Exercise 5 Your next task is to simulate and plot the correlated time series

$$x_{t+1} = \alpha \, x_t + \epsilon_{t+1} \quad \text{where} \quad x_0 = 0 \quad \text{and} \quad t = 0, \ldots, T$$

The sequence of shocks $\{\epsilon_t\}$ is assumed to be iid and standard normal

In your solution, restrict your import statements to

```
from pylab import plot, show
from random import normalvariate
```

Set $T = 200$ and $\alpha = 0.9$

Solution: *View solution*

Exercise 6 To do the next exercise, you will need to know how to produce a plot legend

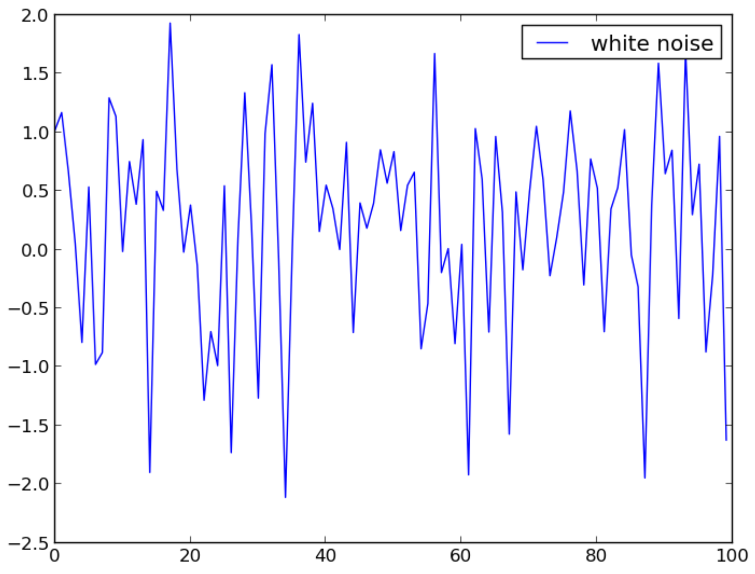
The following example should be sufficient to convey the idea

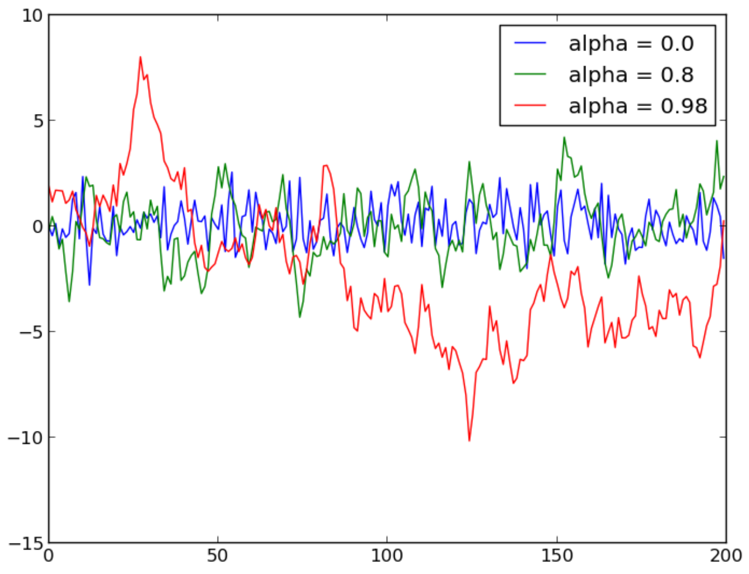
```
from pylab import plot, show, legend
from random import normalvariate
x = [normalvariate(0, 1) for i in range(100)]
plot(x, 'b-', label="white noise")
legend()
show()
```

Running it produces a figure like so

Now, starting with your solution to exercise 5, plot three simulated time series, one for each of the cases $\alpha = 0$, $\alpha = 0.8$ and $\alpha = 0.98$

In particular, you should produce (modulo randomness) a figure that looks as follows


white noise
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0
20
40
60
80
100


alpha = 0.0
alpha = 0.8
alpha = 0.98
10
5
0
−5
−10
−15
0
50
100
150
200

In your solution, please restrict your import statements to

```
from pylab import plot, show, legend
from random import normalvariate
```

Also, use a `for` loop to step through the α values

Important hints:

- If you call the `plot()` function multiple times before calling `show()`, all of the lines you produce will end up on the same figure
  - And if you omit the argument `'b-'` to the plot function, pylab will automatically select different colors for each line
- The expression `'foo' + str(42)` evaluates to `'foo42'`

Solution: *View solution*

Python Essentials

In this lecture we'll cover features of the language that are essential to reading and writing Python

Overview of This Lecture

Topics:

- Some important data types that we haven't covered yet
- Basic file I/O
- The Pythonic approach to iteration
- More on user-defined functions
- Comparisons and logic
- Standard Python style

More Data Types

So far we've met several common data types, including strings, integers, floats and lists

Let's review some other common types

Primitive Data Types One very simple data type is Boolean values, which can be either `True` or `False`

```
>>> x = True
>>> y = 100 < 10   # Python evaluates expression on right and assigns it to y
>>> y
False
>>> type(y)
<type 'bool'>
```

In arithmetic expressions, `True` is converted to `1` and `False` is converted `0`

```
>>> x + y
1
>>> x * y
0
>>> True + True
2
>>> bools = [True, True, False, True]  # List of Boolean values
>>> sum(bools)
3
```

This is called *Boolean arithmetic*, and we will use it a great deal

Complex numbers are another primitive data type in Python

```
>>> x = complex(1, 2)
>>> y = complex(2, 1)
>>> x * y
5j
```

There are several more primitive data types that we'll introduce as necessary

Containers Python has several basic types for storing collections of (possibly heterogeneous) data

We have already discussed lists

A related data type is *tuples*, which are "immutable" lists

```
>>> x = ('a', 'b')  # Round brackets instead of the square brackets used for lists
>>> x = 'a', 'b'    # Or no brackets at all---the meaning is identical
>>> x
('a', 'b')
>>> type(x)
<type 'tuple'>
```

In Python, an object is called "immutable" if, once created, the object cannot be changed

Lists are mutable while tuples are not

```
>>> x = [1, 2]  # Lists are mutable
>>> x[0] = 10   # Now x = [10, 2], so the list has "mutated"
>>> x = (1, 2)  # Tuples are immutable
>>> x[0] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

We'll say more about mutable vs immutable a bit later, and explain why the distinction is important

Tuples (and lists) can be "unpacked" as follows

```
>>> integers = (10, 20, 30)
>>> x, y, z = integers
>>> x
10
>>> y
20
```

You've actually *seen an example of this* already

Tuple unpacking is convenient and we'll use it often

Two other container types we should mention before moving on are sets and dictionaries

Dictionaries are much like lists, except that the items are named instead of numbered

```
>>> d = {'name': 'Frodo', 'age': 33}
>>> type(d)
<type 'dict'>
>>> d['age']
33
```

The names `'name'` and `'age'` are called the *keys*

The objects that the keys are mapped to (`'Frodo'` and `33`) are called the `values`

Sets are unordered collections without duplicates, and set methods provide the usual set theoretic operations

```
>>> s1 = {'a', 'b'}
>>> type(s1)
<type 'set'>
>>> s2 = {'b', 'c'}
>>> s1.issubset(s2)
False
>>> s1.intersection(s2)
set(['b'])
```

The `set()` function creates sets from sequences

```
>>> s3 = set(('foo', 'bar', 'foo'))
>>> s3
set(['foo', 'bar'])  # Unique elements only
```

Input and Output

Let's have a quick look at basic file input and output

We discuss only reading and writing to text files

Input and Output Let's start with writing

```
>>> f = open('newfile.txt', 'w')    # Open 'newfile.txt' for writing
>>> f.write('Testing\n')            # Here '\n' means new line
>>> f.write('Testing again')
>>> f.close()
```

Here

- The built-in function `open()` creates a file object for writing to
- Both `write()` and `close()` are methods of file objects

Where is this file that we've created?

Recall that Python maintains a concept of the current working directory (cwd), which can be obtained by

```
import os
print os.getcwd()
```

(In the IPython notebook, `pwd` should also work)

If a path is not specified, then this is where Python writes to

You can confirm that the file `newfile.txt` is in your cwd using a file browser or some other method

(In IPython, use `ls` to list the files in the cwd)

We can also use Python to read the contents of `newline.txt` as follows

```
>>> f = open('newfile.txt', 'r')
>>> out = f.read()
>>> out
'Testing\nTesting again'
>>> print out
Testing
Testing again
```

Paths Note that if `newfile.txt` is not in the cwd then this call to `open()` fails

In this case you can either specify the full path to the file

```
>>> f = open('insert_full_path_to_file/newfile.txt', 'r')
```

or change the current working directory to the location of the file via `os.chdir('path_to_file')`

(In IPython, use `cd` to change directories)

Details are OS specific, by a Google search on paths and Python should yield plenty of examples

Iterating

One of the most important tasks in computing is stepping through a sequence of data and performing a given action

One of Python's strengths is its simple, flexible interface to this kind of iteration via the `for` loop

Looping over Different Objects Many Python objects are "iterable", in the sense that they can be placed to the right of `in` within a `for` loop statement

To give an example, suppose that we have a file called `us_cities.txt` listing US cities and their population

```
new york: 8244910
los angeles: 3819702
chicago: 2707120
houston: 2145146
philadelphia: 1536471
phoenix: 1469471
san antonio: 1359758
san diego: 1326179
dallas: 1223229
```

Suppose that we want to make the information more readable, by capitalizing names and adding commas to mark thousands

The following program reads the data in and makes the conversion

```
data_file = open('us_cities.txt', 'r')
for line in data_file:
    city, population = line.split(':')            # Tuple unpacking
    city = city.title()                            # Capitalize city names
    population = '{0:,}'.format(int(population))  # Add commas to numbers
    print(city.ljust(15) + population)
data_file.close()
```

Here `format()` is a powerful string method used for inserting variables into strings

The output is as follows

```
New York: 8,244,910
Los Angeles: 3,819,702
Chicago: 2,707,120
Houston: 2,145,146
Philadelphia: 1,536,471
Phoenix: 1,469,471
San Antonio: 1,359,758
San Diego: 1,326,179
Dallas: 1,223,229
```

The reformatting of each line is the result of three different string methods, the details of which can be left till later

The interesting part of this program for us is line 2, which shows that

1. The file object `f` is iterable, in the sense that it can be placed to the right of `in` within a `for` loop
2. Iteration steps through each line in the file

This leads to the clean, convenient syntax shown in our program

Many other kinds of objects are iterable, and we'll discuss some of them later on

Looping without Indices One thing you might have noticed is that Python tends to favor looping without explicit indexing

For example,

```
for x in x_values:
    print x * x
```

is preferred to

```
for i in range(len(x_values)):
    print x_values[i] * x_values[i]
```

When you compare these two alternatives, you can see why the first one is preferred

Python provides some facilities to simplify looping without indices

One is `zip()`, which is used for stepping through pairs from two sequences

For example, try running the following code

```
countries = ('Japan', 'Korea', 'China')
cities = ('Tokyo', 'Seoul', 'Beijing')
for country, city in zip(countries, cities):
    print 'The capital of {0} is {1}'.format(country, city)
```

If we actually need the index from a list, one option is to use `enumerate()`

To understand what `enumerate()` does, consider the following example

```
letter_list = ['a', 'b', 'c']
for index, letter in enumerate(letter_list):
    print "letter_list[{0}] = '{1}'".format(index, letter)
```

The output of the loop is

```
letter_list[0] = 'a'
letter_list[1] = 'b'
letter_list[2] = 'c'
```

Comparisons and Logical Operators

Comparisons Many different kinds of expressions evaluate to one of the Boolean values (i.e., `True` or `False`)

A common type is comparisons, such as

```
>>> x, y = 1, 2
>>> x < y
True
>>> x > y
False
```

One of the nice features of Python is that we can *chain* inequalities

```
>>> 1 < 2 < 3
True
>>> 1 <= 2 <= 3
True
```

As we saw earlier, when testing for equality we use ==

```
>>> x = 1    # Assignment
>>> x == 2   # Comparison
False
```

For "not equal" use `!=`

```
>>> 1 != 2
True
```

We also note that when testing conditions, we can use any expression

```
>>> x = 'yes' if 42 else 'no'
>>> x
'yes'
>>> x = 'yes' if [] else 'no'
>>> x
'no'
```

The rule is

- `False`: Expressions that evaluate to 0 or 0.0, empty sequences (strings, lists, etc.), `None`
- `True`: (Almost) all other values

Combining Expressions We can combine expressions using `and`, `or` and `not`

These are the standard logical connectives (conjunction, disjunction and denial)

```
>>> 1 < 2 and 'f' in 'foo'
True
>>> 1 < 2 and 'g' in 'foo'
False
>>> 1 < 2 or 'g' in 'foo'
True
```

```
>>> not True
False
>>> not not True
True
```

Remember

- `P and Q` is `True` if both are `True`, else `False`
- `P or Q` is `False` if both are `False`, else `True`

More Functions

Let's talk a bit more about functions, which are all-important for good programming style

Python has a number of built-in functions that are available without `import`

We have already met some of them

```
>>> max(19, 20)
20
>>> range(4)
[0, 1, 2, 3]
>>> str(22)
'22'
>>> type(22)
<type 'int'>
```

Two more useful built-in functions are `any()` and `all()`

```
>>> bools = False, True, True
>>> all(bools)   # True if all are True and False otherwise
False
>>> any(bools)   # False if all are False and True otherwise
True
```

The full list of Python built-ins is here

Now let's talk some more about user-defined functions constructed using the keyword `def`

Why Write Functions? User defined functions are important for improving the clarity of your code by

- separating different strands of logic
- facilitating code reuse

(Writing the same thing twice is always a bad idea)

The basics of user defined functions were discussed *here <user_defined_functions>*

The Flexibility of Python Functions As we discussed in the *previous lecture*, Python functions are very flexible

In particular

- Any number of functions can be defined in a given file
- Any object can be passed to a function as an argument, including other functions
- Functions can be (and often are) defined inside other functions
- A function can return any kind of object, including functions

We already *gave an example* of how straightforward it is to pass a function to a function

Note that a function can have arbitrarily many `return` statements (including zero)

Execution of the function terminates when the first return is hit, allowing code like the following example

```
def f(x):
    if x < 0:
        return 'negative'
    return 'nonnegative'
```

Functions without a return statement automatically return the special Python object `None`

Doc Strings Python has a system for adding comments to functions, modules, etc. called *doc strings*

The nice thing about doc strings is that they are available at run-time

For example, if you run this code

```
def f(x):
    """
    This function squares its argument
    """
    return x**2
```

and then type `help(f)`, the interpreter will print the doc string

- In IPython, `f?` does the same thing, while `f??` adds even more

One-Line Functions: `lambda` The `lambda` keyword is used to create simple functions on one line

For example, the definitions

```
def f(x):
    return x**3
```

and

```
f = lambda x: x**3
```

are entirely equivalent

To see why `lambda` is useful, suppose that we want to calculate $\int_0^2 x^3 dx$ (and have forgotten our high-school calculus)

The SciPy library has a function called `quad` that will do this calculation for us

The syntax of the `quad` function is `quad(f, a, b)` where `f` is a function and `a` and `b` are numbers

To create the function $f(x) = x^3$ we can use `lambda` as follows

```
>>> from scipy.integrate import quad
>>> quad(lambda x: x**3, 0, 2)
(4.0, 4.440892098500626e-14)
```

Here the function created by `lambda` is said to be *anonymous*, because it was never given a name

Keyword Arguments If you did the exercises in the *previous lecture*, you would have come across the statement

```
plot(x, 'b-', label="white noise")
```

In this call to Pylab's `plot` function, notice that the last argument is passed in `name=argument` syntax

This is called a *keyword argument*, with `label` being the keyword

Non-keyword arguments are called *positional arguments*, since their meaning is determined by order

- `plot(x, 'b-', label="white noise")` is different to `plot('b-', x, label="white noise")`

Keyword arguments are particularly useful when a function has a lot of arguments, in which case it's hard to remember the right order

You can adopt keyword arguments in user defined functions with no difficulty

The next example illustrates the syntax

```
def f(x, coefficients=(1, 1)):
    a, b = coefficients
    return a + b * x
```

After running this code we can call it as follows

```
>>> f(2, coefficients=(0, 0))
0
>>> f(2)   # Use default values (1, 1)
3
```

Notice that the keyword argument values we supplied in the definition of `f` become the default values

Coding Style and PEP8

To learn more about the Python programming philosophy type `import this` at the prompt

Among other things, Python strongly favors consistency in programming style

We've all heard the saying about consistency and little minds

In programming, as in mathematics, quite the opposite is true

- A mathematical paper where the symbols $\cup$ and $\cap$ were reversed would be very hard to read, even if the author told you so on the first page

In Python, the style which all good programs follow is set out in PEP8

We recommend that you slowly learn it, and following it in your programs

Exercises

Exercise 1 Part 1: Given two numeric lists or tuples `x_vals` and `y_vals` of equal length, compute their inner product using `zip()`

Part 2: In one line, count the number of even numbers in 0,...,99

- Hint: `x % 2` returns 0 if `x` is even, 1 otherwise

Part 3: Given `pairs = ((2, 5), (4, 2), (9, 8), (12, 10))`, count the number of pairs `(a, b)` such that both `a` and `b` are even

Solution: *View solution*

Exercise 2 Consider the polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n = \sum_{i=0}^{n} a_i x^i \tag{1.1}$$

Write a function `p` such that `p(x, coeff)` that computes the value in (1.1) given a point `x` and a list of coefficients `coeff`

Try to use `enumerate()` in your loop

Solution: *View solution*

Exercise 3 Write a function which takes a string as an argument and returns the number of capital letters in the string

Hint: `'foo'.upper()` returns `'FOO'`

Solution: *View solution*

Exercise 4 Write a function which takes two sequences `seq_a` and `seq_b` as arguments and returns `True` if every element in `seq_a` is also an element of `seq_b`, else `False`

- By "sequence" we mean a list, a tuple or a string
- Do the exercise without using sets and set methods

Solution: *View solution*

Exercise 5 When we cover the numerical libraries, we will see they include many alternatives for interpolation and function approximation

Nevertheless, it's a nice exercise to write our own

To this end, without using any imports, write a function `linapprox` that takes as arguments

- A function `f` mapping some interval $[a, b]$ into $\mathbb{R}$
- two scalars `a` and `b` providing the limits of this interval
- An integer `n` determining the number of grid points
- A number `x` satisfying `a <= x <= b`

and returns the piecewise linear interpolation of `f` at `x`, based on `n` evenly spaced grid points `a = point[0] < point[1] < ... < point[n-1] = b`

Aim for clarity, not efficiency

Solution: *View solution*

Object Oriented Programming

Overview of This Lecture

OOP is one of the major paradigms in programming, and nicely supported in Python

Python is not purely object oriented, but OOP is certainly a significant part of Python

OOP is all about how to **organize** your code

Proper organization of code is a critical determinant of productivity

(Without meaning to belittle, if you think that the topic of organizing code sounds inconsequential, that could be because you haven't done a great deal of programming)

In any case, OOP is a part of Python, and to progress further it's necessary to understand the basics

About OOP

Let's start with a general review of OOP, and then progress to Python specifics

Key Concepts The traditional (non-OOP) paradigm is called *procedural*, and works as follows

- The program has a state, which is the values of its variables
- Functions are called to act on this data according to the task
- Data is passed backwards and forwards via function calls

In contrast, in the OOP paradigm, data and functions are bundled together into "objects"

An example is a Python list, which not only stores data, but also knows how to sort itself, etc.

```
>>> x = [1, 5, 4]
>>> x.sort()
>>> x
[1, 4, 5]
```

In the OOP setting, functions are usually called *methods* (e.g., `sort` is a list method)

Common programming languages:

- Python supports both procedural and object-oriented programming
- C is a procedural language
- C++ is is C with OOP added on top
- Fortran and MATLAB are mainly procedural, but with some OOP recently tacked on
- JAVA and Ruby are relatively pure OOP

OOP has become an important concept in modern software engineering because

- It can help facilitate clean, efficient code (when used well)
- The OOP design pattern fits well with the human brain

Standard Terminology A *class definition* is a blueprint for a particular class of objects (e.g., strings, or complex numbers), describing

- what kind of data it stores
- what methods it has for acting on this data

An *object* or *instance* is a realization of the class, created from the blueprint

- Each instance has its own unique data
- Methods set out in the class definition act on this (and other) data

In Python, the data and methods of an object are collectively referred to as *attributes*

Attributes are accessed via "dotted attribute notation"

- `object_name.data`
- `object_name.method_name()`

In the example

```
>>> x = [1, 5, 4]
>>> x.sort()
>>> x.__class__
<type 'list'>
```

- `x` is an object or instance, created from the definition for Python lists, but with its own particular data
- `x.sort()` and `x.__class__` are two attributes of `x`
- `dir(x)` can be used to view all the attributes of `x`

Why is it Useful? OOP is useful for the same reason that abstraction is useful: for recognizing and organizing common phenomena

- E.g., abstracting certain asymmetric information problems leads to the theory of principles and agents

For an example more relevant to OOP, consider the open windows on your desktop

Windows have common functionality and individual data, which makes them suitable for implementing with OOP

- individual data: contents of specific windows
- common functionality: closing, maximizing, etc.

Your window manager almost certainly uses OOP to generate and manage these windows

- individual windows created as objects / instances from a class definition, with their own data
- common functionality implemented as methods, which all of these objects share

Another, more prosaic, use of OOP is data encapsulation

Data encapsulation means storing variables inside some structure so that they are not directly accessible

The alternative to this is filling the global namespace with variable names, frequently leading to conflicts

- Think of the global namespace as any name you can refer to without a dot in front of it

For example, the modules `os` and `sys` both define a different attribute called `path`

The following code leads immediately to a conflict

```
from os import path
from sys import path
```

At this point, both variables have been brought into the global namespace, and the second will shadow the first

A better idea is to replace the above with

```
import os
import sys
```

and then reference the `path` you want with either `os.path` or `sys.path`

In this example, we see that modules provide one means of data encapsulation

As will now become clear, OOP provides another

Defining Your Own Classes

As a first step we are going to try defining very simple classes, the main purpose of which is data encapsulation

Suppose that we are solving an economic model, and one small part of the model is a firm, characterized by

- a production function $f(k) = k^{0.5}$
- a discount factor $\beta = 0.99$
- a borrowing constraint $\kappa = 10$

One option is to declare all these as global variables

A nicer one is to have a single `firm` object as the global variable, and f, β, κ accessible as

- `firm.f`, `firm.beta`, `firm.kappa`

We can do this very easily by running the following code

```python
class Firm:
    pass                # In Python, "pass" essentially means do nothing
firm = Firm()
firm.f = lambda k: k**0.5
firm.beta = 0.99
firm.kappa = 10
```

Here

- The first two lines form the simplest class definition possible in Python—in this case called `Firm`
- The third line creates an object called `firm` as an instance of class `Firm`
- The last three lines dynamically add attributes to the object `firm`

Data and Methods The `Firm` example is only barely OOP—in fact you can do the same kind of think with a MATLAB or C struct

Usually classes also define methods that act on the data contained by the object

- For example, the list method `sort()` in `x.sort()`

Let's try to build something a bit closer to this standard conception of OOP

Since the notation used to define classes seems complex on first pass, we will start with a very simple (and rather contrived) example

In particular, let's build a class to represent dice

The data associated with a given dice will be the side facing up

The only method will be a method to roll the dice (and hence change the state)

The following is **psuedocode**—a class definition in a mix of Python and plain English

```
class Dice:

    data:
        current_face -- the side facing up (i.e., number of dots showing)

    methods:
        roll -- roll the dice (i.e., change current_face)
```

Here's actual Python code, in file `dice.py`

```
import random

class Dice:

    faces = (1, 2, 3, 4, 5, 6)

    def __init__(self):
       self.current_face = 1

    def roll(self):
        self.current_face = random.choice(Dice.faces)
```

There's some difficult notation here, but the broad picture is as follows:

- The `faces` variable is a *class attribute*
    - it will be shared by every member of the class (i.e., every dice)
- The `current_face` variable is an *instance attribute*
    - each dice we create will have its own version
- The `__init__` method is a special method called a *constructor*
    - Used to create instances (objects) from the class definition, with their own data

The `roll` method rolls the dice, changing the state of a particular instance

Once we've run the program, the class definition is loaded into memory

```
>>> Dice
<class __main__.Dice at 0x83de0bc>
>>> dir(Dice)
['__doc__', '__init__', '__module__', 'faces', 'roll']
>>> Dice.faces
(1, 2, 3, 4, 5, 6)
```

Let's now create two die

```
>>> d = Dice()
>>> e = Dice()
```

These two statements implicitly call the `__init__` method to build two instances of `Dice`

When we roll each dice, the `roll` method will only affect the instance variable of that particular instance

```
>>> d.roll()
>>> d.current_face
2
>>> e.roll()
>>> e.current_face
5
```

The most difficult part of all of this notation is the `self` keyword

The simplest way to think of it is that `self` refers to a particular instance

If we want to refer to instance variables, as opposed to class or global variables, then we need to use `self`

In addition, we need to put `self` as the first argument to every method defined in the class

Further Details You might want to leave it at that for now, but if you still want to know more about `self`, here goes:

Consider the method call `d.roll()`

This is in fact translated by Python into to the call `Dice.roll(d)`

So in fact we are calling method `roll()` defined in class object `Dice` with instance `d` as the argument

Hence, when `roll()` executes, `self` is bound to `d`

In this way, `self.current_face = random.choice(faces)` affects `d.current_face`, which is what we want

Example 2: The Quadratic Map Let's look at one more example

The quadratic map difference equation is given by

$$x_{t+1} = 4(1 - x_t)x_t, \quad x_0 \in [0, 1] \text{ given} \tag{1.2}$$

Let's write a class for generating time series, where the data is the current location of the state x_t

Here's one implementation, in file `quadmap_class.py`

```python
class QuadMap:

    def __init__(self, initial_state):
        self.x = initial_state

    def update(self):
        "Apply the quadratic map to update the state."
        self.x = 4 * self.x * (1 - self.x)

    def generate_series(self, n):
        """
        Generate and return a trajectory of length n, starting at the
        current state.
        """
        trajectory = []
        for i in range(n):
            trajectory.append(self.x)
            self.update()
        return trajectory
```

Here's an example of usage, after running the code

```python
>>> q = QuadMap(0.2)
>>> q.x
0.2
>>> q.update()
>>> q.x
0.64000000000000012
>>> q.generate_series(3)
[0.64000000000000012, 0.92159999999999986, 0.28901376000000045]
```

Special Methods

Python provides certain special methods with which a number of neat tricks can be performed

For example, recall that lists and tuples have a notion of length, and this length can be queried via the `len` function

```
>>> x = (10, 20)
>>> len(x)
2
```

If you want to provide a return value for the `len` function when applied to your user-defined object, use the `__len__` special method

```
class Foo:

    def __len__(self):
        return 42
```

Now we get

```
>>> f = Foo()
>>> len(f)
>>> 42
```

A special method we will use regularly is the `__call__` method

This method can be used to make your instances callable, just like functions

```
class Foo:

    def __call__(self, x):
        return x + 42
```

After running we get

```
>>> f = Foo()
>>> f(8)
>>> 50
```

Exercise 1 provides a more useful example

Exercises

Exercise 1 The empirical distribution function corresponding to a sample $\{X_i\}_{i=1}^n$ is defined as

$$F_n(x) := \frac{1}{n} \sum_{i=1}^n \mathbf{1}\{X_i \leq x\} \qquad (x \in \mathbb{R}) \tag{1.3}$$

Here $\mathbf{1}\{X_i \leq x\}$ is an indicator function (one if $X_i \leq x$ and zero otherwise) and hence $F_n(x)$ is the fraction of the sample which falls below x

The Glivenko–Cantelli Theorem states that, provided that the sample is iid, the ecdf F_n converges to the true distribution function F

Implement F_n as a class called `ecdf`, where

- A given sample $\{X_i\}_{i=1}^n$ is the instance data, stored as `self.observations`
- The class implements a `__call__` method, which returns $F_n(x)$ for any x

Your code should work as follows (modulo randomness)

```
>>> from random import uniform
>>> samples = [uniform(0, 1) for i in range(10)]
>>> F = ecdf(samples)
>>> F(0.5)  # Evaluate ecdf at x = 0.5
>>> 0.29
>>> F.observations = [uniform(0, 1) for i in range(1000)]
>>> F(0.5)
>>> 0.479
```

Solution: *View solution*

Exercise 2 Build a simple class to represent and manipulate polynomial functions, without using any `import` statements

The data for the class will be the coefficients, which define a unique polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n = \sum_{n=0}^{N} a_n x^n \qquad (x \in \mathbb{R}) \tag{1.4}$$

Provide two methods that

- Evaluate the polynomial (1.4), returning $p(x)$ for any x
- Differentiate the polynomial, replacing original coefficients with those of its derivative p'

Solution: *View solution*

How it Works: Data, Variables and Names

Overview of This Lecture

The objective of the lecture is to provide deeper understanding of Python's execution model

Understanding these details is important for writing larger programs

You should feel free to **skip this material on first pass** and continue on to the applications

We provide this material mainly as a reference, and for returning to occasionally to build your Python skills

Objects

We discussed objects briefly in *the previous lecture*

Objects are usually thought of as instances of some class definition, typically combining both data and methods (functions)

For example

```
>>> x = ['foo', 'bar']
```

creates (an instance of) a list, possessing various methods (`append`, `pop`, etc.)

In Python everything in memory is treated as an object

This includes not just lists, strings, etc., but also less obvious things, such as

- functions (once they have been read in to memory)

- modules (ditto)
- files opened for reading or writing
- integers, etc.

At this point it is helpful to have a clearer idea of what an object is in Python

In Python, an *object* is a collection of data and instructions held in computer memory that consists of

1. a type
2. some content
3. a unique identity
4. zero or more methods

These concepts are discussed sequentially in the remainder of this section

Type Python understands and provides for different types of objects, to accommodate different types of data

The type of an object can be queried via `type(object_name)`

For example

```
>>> s = 'This is a string'
>>> type(s)
<type 'str'>
>>> x = 42   # Now let's create an integer
>>> type(x)
<type 'int'>
```

The type of an object matters for many expressions

For example, the addition operator between two strings means concatenation

```
>>> '300' + 'cc'
'300cc'
```

On the other hand, between two numbers it means ordinary addition

```
>>> 300 + 400
700
```

Consider the following expression

```
>>> '300' + 400
```

Here we are mixing types, and it's unclear to Python whether the user wants to

- convert `'300'` to an integer and then add it to `400`, or
- convert `400` to string and then concatenate it with `'300'`

Some languages might try to guess, by Python is *strongly typed*

- Type is important, and implicit type conversion is rare
- Python will respond instead by raising a `TypeError`

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

To avoid the error, you need to clarify by changing the relevant type

For example,

```
>>> int('300') + 400   # To add as numbers, change the string to an integer
700
```

Content The content of an object seems like an obvious concept

For example, if we set `x = 42` then it might seem that the content of `x` is just the number 42

But actually, there's more, as the following example shows

```
>>> x = 42
>>> x
42
>>> x.imag
0
>>> x.__class__
<type 'int'>
```

When Python creates this integer object, it stores with it various auxiliary information, such as the imaginary part, and the type

As discussed *previously*, any name following a dot is called an *attribute* of the object to the left of the dot

- For example, `imag` and `__class__` are attributes of `x`

Identity In Python, each object has a unique identifier, which helps Python (and us) keep track of the object

The identity of an object can be obtained via the `id()` function

```
>>> y = 2.5
>>> z = 2.5
>>> id(y)
166719660
>>> id(z)
166719740
```

In this example, `y` and `z` happen to have the same value (i.e., `2.5`), but they are not the same object

The identity of an object is in fact just the address of the object in memory

Methods As discussed earlier, methods are functions that are bundled with objects

Formally, methods are attributes of objects that are callable (i.e., can be called as functions)

```
>>> x = ['foo', 'bar']
>>> callable(x.append)
True
>>> callable(x.__doc__)
False
>>>
```

Methods typically act on the data contained in the object they belong to, or combine that data with other data

```
>>> x = ['a', 'b']
>>> x.append('c')
>>> s = 'This is a string'
>>> s.upper()
```

```
'THIS IS A STRING'
>>> s.lower()
'this is a string'
>>> s.replace('This', 'That')
'That is a string'
```

A great deal of Python functionality is organized around method calls

For example, consider the following piece of code

```
>>> x = ['a', 'b']
>>> x[0] = 'aa'  # Item assignment using square bracket notation
>>> x
['aa', 'b']
```

It doesn't look like there are any methods used here, but in fact the square bracket assignment notation is just a neat interface to a method call

What actually happens is that Python calls the `__setitem__` method, as follows

```
>>> x = ['a', 'b']
>>> x.__setitem__(0, 'aa')  # Equivalent to x[0] = 'aa'
>>> x
['aa', 'b']
```

(If you wanted to you could change the `__setitem__` method, so that square bracket assignment does something totally different)

Everything is an Object Above we said that in Python everything is an object—let's look at this again

Consider, for example, functions

When Python reads a function definition, it creates a function object and stores it in memory

The following code illustrates

```
>>> def f(x): return x**2
...
>>> f
<function f at 0xb73ebd4c>
>>> type(f)
<type 'function'>
>>> id(f)
3074342220L
>>> f.func_name
'f'
```

We can see that `f` has type, identity, attributes and so on—just like any other object

Likewise modules loaded into memory are treated as objects

```
>>> import math
>>> id(math)
3074329380L
```

This uniform treatment of data in Python (everything is an object) helps keep the language simple and consistent

Iterables and Iterators

We *already said something* about iterating in Python

Let's learn a bit more about how the Python `for` loop works

Iterators are a uniform interface to stepping through elements in a collection

Here we'll talk about using iterators—later we'll learn how to build our own

Iterators Formally, an *iterator* is an object with a `next()` method

For example, file objects are iterators

To see this, let's have another look at the *US cities data*

```
>>> f = open('us_cities.txt', 'r')
>>> f.next()
'new york: 8244910\n'
>>> f.next()
'los angeles: 3819702\n'
```

We see that file objects do indeed have `next` methods, and calling that method returns the next line in the file

The objects returned by `enumerate()` are also iterators

```
>>> e = enumerate(['foo', 'bar'])
>>> e.next()
(0, 'foo')
>>> e.next()
(1, 'bar')
```

as are the reader objects from the `csv` module

```
>>> from csv import reader
>>> f = open('test_table.csv', 'r')
>>> nikkei_data = reader(f)
>>> nikkei_data.next()
['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close']
>>> nikkei_data.next()
['2008-05-19', '14294.52', '14343.19', '14219.08', '14269.61', '133800', '14269.61']
```

or objects returned by `urllib.urlopen()`

```
>>> import urllib
>>> webpage = urllib.urlopen("http://www.cnn.com")
>>> webpage.next()
'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN""http://www.w3.org/...' # etc
>>> webpage.next()
'<meta http-equiv="refresh" content="1800;url=?refresh=1">\n'
>>> webpage.next()
'<meta name="Description" content="CNN.com delivers the latest breaking news and information..' # et
```

Iterators in For Loops All iterators can be placed to the right of the `in` keyword in `for` loop statements

In fact this is how the `for` loop works: If we write

```
for x in iterator:
    <code block>
```

then the interpreter

- calls `iterator.next()` and binds `x` to the result
- executes the code block

- repeats until a `StopIteration` error occurs

So now you know how this magical looking syntax works

```
f = open('somefile.txt', 'r')
for line in f:
    # do something
```

The interpreter just keeps

1. calling `f.next()` and binding `line` to the result
2. executing the body of the loop

This continues until a `StopIteration` error occurs

Iterables You already know that we can put a Python list to the right of `in` in a `for` loop

```
for i in range(2):
    print 'foo'
```

So does that mean that a list is an iterator?

The answer is no:

```
>>> type(x)
<type 'list'>
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'next'
```

So why can we iterate over a list in a `for` loop?

The reason is that a list is *iterable* (as opposed to an iterator)

Formally, an object is iterable if it can be converted to an iterator using the built-in function `iter()`

Lists are one such object

```
>>> x = ['foo', 'bar']
>>> type(x)
<type 'list'>
>>> y = iter(x)
>>> type(y)
<type 'listiterator'>
>>> y.next()
'foo'
>>> y.next()
'bar'
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Many other objects are iterable, such as dictionaries and tuples

Of course, not all objects are iterable

```
>>> iter(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

To conclude our discussion of `for` loops

- `for` loops work on either iterators or iterables
- In the second case, the iterable is converted into an iterator before the loop starts

Iterators and built-ins Some built-in functions that act on sequences also work with iterables

- `max()`, `min()`, `sum()`, `all()`, `any()`

For example

```
>>> x = [10, -10]
>>> max(x)
10
>>> y = iter(x)
>>> type(y)
<type 'listiterator'>
>>> max(y)
10
```

One thing to remember about iterators is that they are depleted by use

```
>>> x = [10, -10]
>>> y = iter(x)
>>> max(y)
10
>>> max(y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: max() arg is an empty sequence
```

Names and Name Resolution

Variable Names in Python Consider the Python statement

```
>>> x = 42
```

We now know that when this statement is executed, Python creates an object of type `int` in your computer's memory, containing

- the value `42`
- some associated attributes

But what is `x` itself?

In Python, `x` is called a *name*, and the statement

```
>>> x = 42
```

binds the name `x` to the integer object we have just discussed

Under the hood, this process of binding names to objects is implemented as a dictionary—more about this in a moment

There is no problem binding two or more names to the one object, regardless of what that object is

```
>>> def f(string):      # Create a function called f
...     print(string)   # that prints any string it's passed
...
>>> g = f
>>> id(g) == id(f)
True
>>> g('test')
test
```

In the first step, a function object is created, and the name `f` is bound to it

After binding the name `g` to the same object, we can use it anywhere we would use `f`

What happens when the number of names bound to an object goes to zero?

Here's an example of this situation, where the name `x` is first bound to one object and the rebound to another

```
>>> x = 42
>>> id(x)
164994764
>>> x = 'foo'   # No names bound to object 164994764
```

What happens here is that the first object, with identity `164994764` is garbage collected

In other words, the memory slot which stores that object is deallocated, and return to the operating system

Namespaces Recall from the preceding discussion that the statement

```
>>> x = 42
```

binds the name `x` to the integer object on the right-hand side

We also mentioned that this process of binding `x` to the correct object is implemented as a dictionary

This dictionary is called a *namespace*

Definition: A namespace is a symbol table that maps names to objects in memory

Python uses multiple namespaces, creating them on the fly as necessary

For example, every time we import a module, Python creates a namespace for that module

To see this in action, suppose we write a script `math2.py` like this

```
# Filename: math2.py
pi = 'foobar'
```

Now we start the Python interpreter and import it

```
>>> import math2
```

Next let's import the `math` module from the standard library

```
>>> import math
```

Both of these modules have an attribute called `pi`

```
>>> math.pi
3.1415926535897931
>>> math2.pi
'foobar'
```

These two different bindings of `pi` exist in different namespaces, each one implemented as a dictionary

We can look at the dictionary directly, using `module_name.__dict__`

```
>>> import math
>>> math.__dict__
{'pow': <built-in function pow>, ..., 'pi': 3.1415926535897931,...}
>>> import math2
>>> math2.__dict__
{..., '__file__': 'math2.py', 'pi': 'foobar',...}
```

As you know, we access elements of the namespace using the dotted attribute notation

```
>>> math.pi
3.1415926535897931
```

In fact this is entirely equivalent to `math.__dict__['pi']`

```
>>> math.__dict__['pi'] == math.pi
True
```

Viewing Namespaces As we saw above, the `math` namespace can be printed by typing `math.__dict__`

Another way to see its contents is to type `vars(math)`

```
>>> vars(math)
{'pow': <built-in function pow>,...
```

If you just want to see the names, you can type

```
>>> dir(math)
['__doc__', '__name__', 'acos', 'asin', 'atan',...
```

Notice the special names `__doc__` and `__name__`

These are initialized in the namespace when any module is imported

- `__doc__` is the doc string of the module
- `__name__` is the name of the module

```
>>> print math.__doc__
This module is always available.  It provides access to the
mathematical functions defined by the C standard.
>>> math.__name__
'math'
```

Interactive Sessions In Python, all code executed in the interpreter runs in some module

What about commands typed at the prompt >>>?

These are regarded as part of a module called `__main__`

To see the namespace of `__main__` use `vars()` rather than `vars(__main__)`

```
>>> vars()   # After starting Python, before making any assignments
{'__builtins__': <module '__builtin__' (built-in)>,
'__name__': '__main__',
'__doc__': None}
```

Now let's make an assignment

```
>>> x = 4
>>> vars()
{'__builtins__': <module '__builtin__' (built-in)>,
'__name__': '__main__',
'__doc__': None,
'x': 4}
```

The variable `x` has been registered in the namespace

To illustrate further, suppose that we have a script `mod.py` that prints its own `__name__` attribute

```
# Filename: mod.py
print(__name__)
```

Now let's look at two different ways of running it in IPython

```
In [1]: import mod  # Standard import
mod

In [2]: run mod.py  # Run interactively
__main__
```

In the second case, the code is executed as part of `__main__`, so `__name__` is equal to `__main__`

The Global Namespace Python documentation often makes reference to the "global namespace"

Definition The *global namespace* is the namespace of the module currently being executed

For example, suppose that we start the interpreter and begin making assignments

```
>>> x = 3
>>> dir()
['__builtins__', '__doc__', '__name__', 'x']
```

We are now working in the module `__main__`, and hence the namespace for `__main__` is the global namespace

Next, we import a module called `amodule`

```
>>> import amodule
```

At this point, the interpreter creates a namespace for the module `amodule` and starts executing commands in the module

While this occurs, the namespace `amodule.__dict__` is the global namespace

Once execution of the module finishes, the interpreter returns to the module from where the import statement was made

In this case it's `__main__`, so the namespace of `__main__` again becomes the global namespace

Local Namespaces Suppose that inside a module we have access to a function `f`

```
def f(x):
    a = 2
    return a * x
```

Now we call the function

```
y = f(1)
```

When this happens, the interpreter creates a *local namespace* for the function, and registers the variables in that namespace

Variables in the namespace are called *local variables*

After the function returns, the namespace is deallocated and lost

While the function is executing, we can view the contents of the local namespace with `locals()`

```
def f(x):
    a = 2
    print locals()
    return a * x
```

Now we call the function

```
y = f(1)
{'a': 2, 'x': 1}
```

The `__builtins__` Namespace We have been using various built-in functions, such as `max()`, `dir()`, `str()`, `list()`, `len()`, `range()`, `type()`, etc.

How does access to these names work?

- These definitions are stored in a module called `__builtin__`
- They have there own namespace, called `__builtins__`

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> dir(__builtins__)
[... 'iter', 'len', 'license', 'list', 'locals', ...]
```

We can access elements of the namespace as follows

```
>>> __builtins__.max
<built-in function max>
```

But `__builtins__` is special, because we can always access them directly as well

```
>>> max
<built-in function max>
>>> __builtins__.max == max
True
```

The reason why this works is explained in the next section...

Name resolution When we reference a name, how does the Python interpreter find the corresponding value?

At any point of execution, there are either two or three namespaces which can be accessed directly

(Directly means without using a dot, as in `pi` rather than `math.pi`)

If the interpreter is *not* executing a function call then these namespaces are

- The global namespace (of the module being executed)
- The builtin namespace

If we refer to a name such as `x`, the interpreter

- First looks in the global namespace for `x`
- If it's not there, then it looks in the built-in namespace

- If it's not there, it raises a `NameError`

If the interpreter *is* executing a function, then the namespaces are

- The local namespace of the function
- The global namespace (of the module being executed)
- The builtin namespace

Now the interpreter

- First looks in the local namespace
- Then in the global namespace
- Then in the builtin namespace
- If it's not there, it raises a `NameError`

To illustrate this further, consider a script `test.py` that looks as follows

```
def g(x):
    a = 1
    x = x + a
    return x

a = 0
y = g(10)
print "a = ", a, "y = ", y
```

What happens when we run this script?

```
$ python -i test.py
a = 0 , y = 11
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

First,

- The global namespace `{}` is created
- The function object is created, and `g` is bound to it within the global namespace
- The name `a` is bound to `0`, again in the global namespace

Next `g` is called via `y = g(10)`, leading to the following sequence of actions

- The local namespace for the function is created
- Local names `x` and `a` are bound, so that the local namespace becomes `{'x': 10, 'a': 1}`
- Statement `x = x + a` uses the local `a` and local `x` to compute `x + a`, and binds local name `x` to the result
- This value is returned, and `y` is bound to it in the global namespace
- Local `x` and `a` are discarded (and the local namespace is deallocated)

Note that the global `a` was not affected by the local `a`

Mutable Versus Immutable Parameters This is a good time to say a little more about mutable vs immutable objects

Consider the code segment

```
def f(x):
    x = x + 1
    return x
x = 1
print f(x), x
```

We now understand what will happen here: The code prints `2` as the value of `f(x)` and `1` as the value of `x`

First `f` and `x` are registered in the global namespace

The call `f(x)` creates a local namespace and adds `x` to it, bound to `1`

Next, this local `x` is rebound to the new integer object `2`, and this value is returned

None of this affects the global `x`

It's a different story when we use a mutable data type such as a list

```
def f(x):
    x[0] = x[0] + 1
    return x
x = [1]
print f(x), x
```

Prints `[2]` as the value of `f(x)` and *same* for `x`

Here's what happens

- `f` is registered as a function in the global namespace
- `x` bound to `[1]` in the global namespace
- **Call `f(x)`**
    - Creates a local namespace
    - Adds `x` to local namespace, bound to `[1]`
    - The list `[1]` is modified to `[2]`
    - Returns the list `[2]`
    - Local namespace deallocated, local `x` lost
- Global `x` has been modified

Advanced Features

Overview of This Lecture

As with the last lecture, our advice is to **skip this lecture on first pass**, unless you have a burning desire to read it

It's here

1. as a reference, so we can link back to it when required, and
2. for those who have worked through a number of applications, and now want to learn more about the Python language

A variety of topics are treated in the lecture, including generators, exceptions and descriptors

Generators

A generator is a kind of iterator (i.e., it implements a `next()` method)

We will study two ways to build generators: generator expressions and generator functions

Generator Expressions The easiest way to build generators is using *generator expressions*

Just like a list comprehension, but with round brackets

Here is the list comprehension:

```
>>> singular = ('dog', 'cat', 'bird')
>>> type(singular)
<type 'tuple'>
>>> plural = [string + 's' for string in singular]   # Creates a list
>>> plural
['dogs', 'cats', 'birds']
>>> type(plural)
<type 'list'>
```

And here is the generator expression

```
>>> singular = ('dog', 'cat', 'bird')
>>> plural = (string + 's' for string in singular)   # Creates a generator
>>> type(plural)
<type 'generator'>
>>> plural.next()
'dogs'
>>> plural.next()
'cats'
>>> plural.next()
'birds'
```

Since `sum()` can be called on iterators, we can do this

```
>>> sum((x * x for x in range(10)))
285
```

The function `sum()` calls `next()` to get the items, adds successive terms

In fact, we can omit the outer brackets in this case

```
>>> sum(x * x for x in range(10))
285
```

Generator Functions The most flexible way to create generator objects is to use generator functions

Let's look at some examples

Example 1 Here's a very simple example of a generator function

```
def f():
    yield 'start'
    yield 'middle'
    yield 'end'
```

It looks like a function, but uses a keyword `yield` that we haven't met before

Let's see how it works after running this code

```
>>> type(f)           # f itself is a function
<type 'function'>
>>> gen = f()         # Creates a generator object
>>> gen
<generator object at 0xb7cf31ac>
>>> gen.next()
'start'
>>> gen.next()
'middle'
>>> gen.next()
'end'
>>> gen.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

The generator function `f()` is used to create generator objects (in this case `gen`)

Generators are iterators, because they support a `next()` method

The first call to `gen.next()`

- Executes code in the body of `f()` until it meets a `yield` statement
- Returns that value to the caller of `gen.next()`

The second call to `gen.next()` starts executing *from the next line*

```
def f():
    yield 'start'
    yield 'middle'   # This line!
    yield 'end'
```

and continues until the next `yield` statement

At that point it returns the value following `yield` to the caller of `gen.next()`, and so on

When the code block ends, the generator throws a `StopIteration` error

Example 2 Our next example receives an argument `x` from the caller

```
def g(x):
    while x < 100:
        yield x
        x = x * x
```

Let's see how it works

```
>>> g
<function g at 0xb7d6b25c>
>>> gen = g(2)   # Call generator function to make a generator
>>> type(gen)    # gen is an object of type generator
<type 'generator'>
>>> gen.next()
2
>>> gen.next()
4
```

```
>>> gen.next()
16
>>> gen.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

The call `gen = g(2)` binds `gen` to a generator

Inside the generator, the name `x` is bound to `2`

When we call `gen.next()`

- The body of `g()` executes until the line `yield x`, and the value of `x` is returned

Note that value of `x` is retained inside the generator

When we call `gen.next()` again, execution continues *from where it left off*

```
def g(x):
    while x < 100:
        yield x
        x = x * x  # execution continues from here
```

When `x < 100` fails, the generator throws a `StopIteration` error

Note that the loop inside the generator can be infinite

```
def g(x):
    while 1:
        yield x
        x = x * x
```

Advantages of Iterators What's the advantage of using an iterator here?

Suppose we want to sample a binomial(n,0.5)

One way to do it is as follows

```
>>> n = 10000000
>>> draws = [random.uniform(0, 1) < 0.5 for i in range(n)]
>>> sum(draws)
```

But we are creating two huge lists here `range(n)`, and `draws`

This uses lots of memory and is very slow

If we make `n` even bigger then this happens

```
>>> n = 1000000000
>>> draws = [random.uniform(0, 1) < 0.5 for i in range(n)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

We can avoid these problems using iterators

Here is the generator function

```
import random

def f(n):
    i = 1
    while i <= n:
        yield random.uniform(0, 1) < 0.5
        i += 1
```

Now let's do the sum

```
>>> n = 10000000
>>> draws = f(n)
>>> draws
<generator object at 0xb7d8b2cc>
>>> sum(draws)
4999141
```

In summary iterables

- avoid the need to create big lists/tuples, and
- provide a uniform interface to iteration (when can be used transparently in `for` loops)

Handling Exceptions

Runtime errors cause execution of the relevant program to stop

- Frustrating if you're in the middle of a large computation
- Sometimes the debugging information is not very informative
- Disappointing for users of your code

A better way is to add code to your program that deals with errors as they occur

Errors If you've got this far then you've already seen lots of errors

Here's an example

```
>>> def f:
  File "<stdin>", line 1
    def f:
         ^
SyntaxError: invalid syntax
```

This is a syntax error

- Illegal syntax cannot be executed
- Always terminates execution of the program

Here's a different kind of error, unrelated to syntax

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Here's another

```
>>> x = y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

And another

```
>>> 'foo' + 6
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

And another

```
>>> X = []
>>> x = X[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

On each occaision, the interpreter informs us of the error type

- `IndexError`, `ZeroDivisionError`, etc.

In Python, these errors are called *exceptions*

We can catch and deal with exceptions using `try` – `except` blocks

Here's a simple example

```
def f(x):
    try:
        return 1.0 / x
    except ZeroDivisionError:
        print 'Error: division by zero.  Returned None'
    return None
```

When we call f we get the following output

```
>>> f(2)
0.5
>>> f(0)
Error: division by zero.  Returned None
>>> f(0.0)
Error: division by zero.  Returned None
>>>
```

The error is caught and execution of the program is not terminated

Note that other error types are not caught

If we are worried the user might pass in a string, we can catch that error too

```
def f(x):
    try:
        return 1.0 / x
    except ZeroDivisionError:
        print 'Error: Division by zero.  Returned None'
    except TypeError:
        print 'Error: Unsupported operation.  Returned None'
    return None
```

Here's what happens

```
>>> f(2)
0.5
>>> f(0)
Error: Division by zero.  Returned None
```

```
>>> f('foo')
Error: Unsupported operation.  Returned None
>>>
```

If we feel lazy we can catch these errors together

```
def f(x):
    try:
        return 1.0 / x
    except (TypeError, ZeroDivisionError):
        print 'Error: Unsupported operation.  Returned None'
    return None
```

Here's what happens

```
>>> f(2)
0.5
>>> f(0)
Error: Unsupported operation.  Returned None
>>> f('foo')
Error: Unsupported operation.  Returned None
```

If we feel extra lazy we can catch all error types as follows

```
def f(x):
    try:
        return 1.0 / x
    except:
        print 'Error.  Returned None'
    return None
```

In general it's better to be specific

Descriptors

Descriptors solve a common problem regarding management of variables

To understand the issue, consider a `Car` class, performing a variety of tasks that we won't bother to describe

Suppose that this class defines the variables `miles_till_service` and `kms_till_service`, which give the distance until next service in miles and kilometers respectively

A highly simplified version of the class might look as follows

```
class Car(object):

    def __init__(self, miles_till_service=1000):
        self.miles_till_service = miles_till_service
        self.kms_till_service = miles_till_service * 1.61
```

One potential problem we might have here is that a user alters one of these variables but not the other

```
In [2]: car = Car()

In [3]: car.miles_till_service
Out[3]: 1000

In [4]: car.kms_till_service
Out[4]: 1610.0
```

```
In [5]: car.miles_till_service = 6000

In [6]: car.kms_till_service
Out[6]: 1610.0
```

In the last two lines we see that `miles_till_service` and `kms_till_service` are out of sync

What we really want is some mechanism whereby, each time a user sets one of these variables, the other is automatically updated

In Python, this is solved using *descriptors*, an implementation based on which could look as follows

```
class Car(object):

    def __init__(self, miles_till_service=1000):

        self.__miles_till_service = miles_till_service
        self.__kms_till_service = miles_till_service * 1.61

    def set_miles(self, value):
        self.__miles_till_service = value
        self.__kms_till_service = value * 1.61

    def set_kms(self, value):
        self.__kms_till_service = value
        self.__miles_till_service = value / 1.61

    def get_miles(self):
        return self.__miles_till_service

    def get_kms(self):
        return self.__kms_till_service

    miles_till_service = property(get_miles, set_miles)
    kms_till_service = property(get_kms, set_kms)
```

The names `__miles_till_service` and `__kms_till_service` are arbitrary names we are using to store the values of the variables

The objects `miles_till_service` and `kms_till_service` are "properties", use of which invokes the various get and set methods

In any case, we now get the desired behaviour

```
In [8]: car = Car()

In [9]: car.miles_till_service
Out[9]: 1000

In [10]: car.miles_till_service = 6000

In [11]: car.kms_till_service
Out[11]: 9660.0
```

For further information you can refer to the documentation

Recursive Function Calls

This is not something that you will use every day, but it is still useful — you should learn it at some stage

Basically, a recursive function is a function that calls itself

For example, consider the problem of computing x_t for some t when

$$x_{t+1} = 2x_t, \quad x_0 = 1 \tag{1.5}$$

Obviously the answer is 2^t

We can compute this easily enough with a loop

```
def x_loop(t):
    x = 1
    for i in range(t):
        x = 2 * x
    return x
```

We can also use a recursive solution, as follows

```
def x(t):
    if t == 0:
        return 1
    else:
        return 2 * x(t-1)
```

What happens here is that each successive call uses it's own *frame* in the *stack*

- a frame is where the local variables of a given function call are held
- **stack is memory used to process function calls**
    - a First In Last Out (FILO) queue

This example is somewhat contrived, since the first (iterative) solution would usually be preferred to the recursive solution

We'll meet less contrived applications of recursion later on

Exercises

Exercise 1 The Fibonacci numbers are defined by

$$x_{t+1} = x_t + x_{t-1}, \quad x_0 = 0, \; x_1 = 1 \tag{1.6}$$

The first few numbers in the sequence are: `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55`

Write a function to recursively compute the t-th Fibonacci number for any t

Solution: *View solution*

Exercise 2 Complete the following code, and test it using the `test_table.csv` file in the main repository

```
def column_iterator(target_file, column_number):
    """A generator function for CSV files.
    When called with a file name target_file (string) and column number
    column_number (integer), the generator function returns a generator
    which steps through the elements of column column_number in file
    target_file.
    """
    # put your code here

dates = column_iterator('test_table.csv', 1)

for date in dates:
    print date
```

Solution: *View solution*

Exercise 3 Suppose we have a text file `numbers.txt` containing the following lines

```
prices
3
8

7
21
```

Using `try – except`, write a program to read in the contents of the file and sum the numbers, ignoring lines without numbers

Solution: *View solution*

1.3 Part 2: The Scientific Libraries

Some intro

1.3.1 Lectures

NumPy

Overview of This Lecture

NumPy is a first-rate library for numerical programming

- Widely used in academia, finance and industry
- Mature, fast and and stable

In this lecture we introduce the NumPy array data type and fundamental array processing operations

We assume that NumPy is installed on the machine you are using—see *this page* for instructions

Introduction to NumPy

The essential problem that NumPy solves is fast array processing

This is necessary primarily because iteration via loops in interpreted languages (Python, MATLAB, Ruby, etc.) is relative slow

Loops in compiled languages like C and Fortran can be orders of magnitude faster

Why? Because interpreted languages convert commands to machine code and execute them one by one—a difficult process to optimize

Does that mean that we should just switch to C or Fortran for everything?

The answer is no—see *this discussion*

But it does mean that we need libaries like NumPy and SciPy, through which operations can be sent in batches to highly optimized C and Fortran code

NumPy Arrays

The most important thing that NumPy defines is an array data type formally called a `numpy.ndarray`

For example, the `np.zeros` function returns an `numpy.ndarray` of zeros

```
In [1]: import numpy as np

In [2]: a = np.zeros(3)

In [3]: a
Out[3]: array([ 0.,  0.,  0.])

In [4]: type(a)
Out[4]: numpy.ndarray
```

NumPy arrays are somewhat like native Python lists, except that

- Data *must be homogeneous* (all elements of the same type)
- These types must be one of the data types (`dtypes`) provided by NumPy

The most important of these dtypes are:

- float64: 64 bit floating point number
- float32: 32 bit floating point number
- int64: 64 bit integer
- int32: 32 bit integer
- bool: 8 bit True or False

There are also dtypes to represent complex numbers, unsigned integers, etc

On most machines, the default dtype for arrays is `float64`

```
In [7]: a = np.zeros(3)

In [8]: type(a[0])
Out[8]: numpy.float64
```

If we want to use integers we can specify as follows:

```
In [9]: a = np.zeros(3, dtype=int)

In [10]: type(a[0])
Out[10]: numpy.int32
```

Shape and Dimension When we create an array such as

```
In [11]: z = np.zeros(10)
```

`z` is a “flat” array with no dimension— neither row vector nor column vector

The dimension is recorded in the `shape` attribute, which is a tuple

```
In [12]: z.shape
Out[12]: (10,)
```

Here the shape tuple has only one element, which is the length of the array (tuples with one element end with a comma)

To give it dimension, we can change the `shape` attribute

```
In [13]: z.shape = (10, 1)

In [14]: z
Out[14]:
array([[ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.]])

In [15]: z = np.zeros(4)

In [16]: z.shape = (2, 2)

In [17]: z
Out[17]:
array([[ 0.,  0.],
       [ 0.,  0.]])
```

In the last case, to make the 2 by 2 array, we could also pass a tuple to the `zeros()` function, as in `z = np.zeros((2, 2))`

Creating Arrays As we’ve seen, the `np.zeros` function creates an array of zeros

You can probably guess what `np.ones` creates

Related is `np.empty`, which creates arrays in memory that can later be populated with data

```
In [18]: z = np.empty(3)

In [19]: z
Out[19]: array([  8.90030222e-307,   4.94944794e+173,   4.04144187e-262])
```

What are those numbers, by the way?

Here Python allocates 3 contiguous pieces of memory, each with 64 bits, and the existing contents of those memory slots is interpreted as a `float64`

To set up a grid of evenly spaced numbers use `np.linspace`

```
In [20]: z = np.linspace(2, 4, 5)  # From 2 to 4, with 5 elements
```

To create an identity matrix use either `np.identity` or `np.eye`

```
In [21]: z = np.identity(2)

In [22]: z
Out[22]:
array([[ 1.,  0.],
       [ 0.,  1.]])
```

In addition, NumPy arrays can be created from Python lists, tuples, etc. using `np.array`

```
In [23]: z = np.array([10, 20])              # ndarray from Python list

In [24]: z
Out[24]: array([10, 20])

In [25]: type(z)
Out[25]: numpy.ndarray

In [26]: z = np.array((10, 20), dtype=float)    # Here 'float' is equivalent to 'np.float64'

In [27]: z
Out[27]: array([ 10.,  20.])

In [28]: z = np.array([[1, 2], [3, 4]])         # 2D array from a list of lists

In [29]: z
Out[29]:
array([[1, 2],
       [3, 4]])
```

To read in the array data from a text file containing numeric data use `np.loadtxt` or `np.genfromtxt`—see the documentation for details

Array Indexing For a flat array, indexing is the same as Python sequences:

```
In [30]: z = np.linspace(1, 2, 5)

In [31]: z
Out[31]: array([ 1.  ,  1.25,  1.5 ,  1.75,  2.  ])

In [32]: z[0]
Out[32]: 1.0

In [33]: z[0:2]  # Slice numbering is left closed, right open
Out[33]: array([ 1.  ,  1.25])

In [34]: z[-1]
Out[34]: 2.0
```

For 2D arrays the syntax is as follows:

```
In [35]: z = np.array([[1, 2], [3, 4]])

In [36]: z
Out[36]:
array([[1, 2],
       [3, 4]])

In [37]: z[0, 0]
```

```
Out[37]: 1

In [38]: z[0, 1]
Out[38]: 2
```

And so on

Note that indices are still zero-based, to maintain compatibility with Python sequences

Columns and rows can be extracted as follows

```
In [39]: z[0,:]
Out[39]: array([1, 2])

In [40]: z[:,1]
Out[40]: array([2, 4])
```

NumPy arrays of integers can also be used to extract elements

```
In [41]: z = np.linspace(2, 4, 5)

In [42]: z
Out[42]: array([ 2. ,  2.5,  3. ,  3.5,  4. ])

In [43]: indices = np.array((0, 2, 3))

In [44]: z[indices]
Out[44]: array([ 2. ,  3. ,  3.5])
```

Finally, an array of `dtype bool` can be used to extract elements

```
In [45]: z
Out[45]: array([ 2. ,  2.5,  3. ,  3.5,  4. ])

In [46]: d = np.array([0, 1, 1, 0, 0], dtype=bool)

In [47]: d
Out[47]: array([False,  True,  True, False, False], dtype=bool)

In [48]: z[d]
Out[48]: array([ 2.5,  3. ])
```

We'll see why this is useful below

An aside: all elements of an array can be set equal to one number using slice notation

```
In [49]: z = np.empty(3)

In [50]: z
Out[50]: array([ -1.25236750e-041,   0.00000000e+000,   5.45693855e-313])

In [51]: z[:] = 42

In [52]: z
Out[52]: array([ 42.,  42.,  42.])
```

Array Methods Arrays have useful methods, all of which are highly optimized

```
In [53]: A = np.array((4, 3, 2, 1))

In [54]: A
Out[54]: array([4, 3, 2, 1])

In [55]: A.sort()              # Sorts A in place

In [56]: A
Out[56]: array([1, 2, 3, 4])

In [57]: A.sum()               # Sum
Out[57]: 10

In [58]: A.mean()              # Mean
Out[58]: 2.5

In [59]: A.max()               # Max
Out[59]: 4

In [60]: A.argmax()            # Returns the index of the maximal element
Out[60]: 3

In [61]: A.cumsum()            # Cumulative sum of the elements of A
Out[61]: array([ 1,  3,  6, 10])

In [62]: A.cumprod()           # Cumulative product of the elements of A
Out[62]: array([ 1,  2,  6, 24])

In [63]: A.var()               # Variance
Out[63]: 1.25

In [64]: A.std()               # Standard deviation
Out[64]: 1.1180339887498949

In [65]: A.shape = (2, 2)

In [66]: A.T                   # Equivalent to A.transpose()
Out[66]:
array([[1, 3],
       [2, 4]])
```

Another method worth knowing is `searchsorted()`

If `z` is a nondecreasing array, then `z.searchsorted(a)` returns index of first `z` in `z` such that `z >= a`

```
In [67]: z = np.linspace(2, 4, 5)

In [68]: z
Out[68]: array([ 2. ,  2.5,  3. ,  3.5,  4. ])

In [69]: z.searchsorted(2.2)
Out[69]: 1

In [70]: z.searchsorted(2.5)
Out[70]: 1

In [71]: z.searchsorted(2.6)
Out[71]: 2
```

Many of the methods discussed above have equivalent functions in the NumPy namespace

```
In [72]: a = np.array((4, 3, 2, 1))

In [73]: np.sum(a)
Out[73]: 10

In [74]: np.mean(a)
Out[74]: 2.5
```

Operations on Arrays

Algebraic Operations The algebraic operators `+`, `-`, `*`, `/` and `**` all act *elementwise* on arrays

```
In [75]: a = np.array([1, 2, 3, 4])

In [76]: b = np.array([5, 6, 7, 8])

In [77]: a + b
Out[77]: array([ 6,  8, 10, 12])

In [78]: a * b
Out[78]: array([ 5, 12, 21, 32])
```

We can add a scalar to each element as follows

```
In [79]: a + 10
Out[79]: array([11, 12, 13, 14])
```

Scalar multiplication is similar

```
In [81]: a = np.array([1, 2, 3, 4])

In [82]: a * 10
Out[82]: array([10, 20, 30, 40])
```

The two dimensional arrays follow the same general rules

```
In [86]: A = np.ones((2, 2))

In [87]: B = np.ones((2, 2))

In [88]: A + B
Out[88]:
array([[ 2.,  2.],
       [ 2.,  2.]])

In [89]: A + 10
Out[89]:
array([[ 11.,  11.],
       [ 11.,  11.]])

In [90]: A * B
Out[90]:
array([[ 1.,  1.],
       [ 1.,  1.]])
```

In particular, `A * B` is *not* the matrix product, it is an elementwise product

To do matrix multiplication we can either

- convert the arrays into `numpy.matrix` data type
- or use the `np.dot` function

The first case is *discussed below*, for now let's look at `dot`

```
In [137]: A = np.ones((2, 2))

In [138]: B = np.ones((2, 2))

In [139]: np.dot(A, B)
Out[139]:
array([[ 2.,  2.],
       [ 2.,  2.]])
```

With `np.dot` we can also take the inner product of two flat arrays

```
In [91]: A = np.array([1, 2])

In [92]: B = np.array([10, 20])

In [93]: np.dot(A, B)    # Returns a scalar in this case
Out[93]: 50
```

In fact we can use `dot` when one element is a Python list or tuple

```
In [94]: A = np.empty((2, 2))

In [95]: A
Out[95]:
array([[  3.48091887e-262,   1.14802984e-263],
       [  3.61513512e-313,  -1.25232371e-041]])

In [96]: np.dot(A, (0, 1))
Out[96]: array([  1.14802984e-263,  -1.25232371e-041])
```

Here `dot` knows we are postmultiplying, so `(0, 1)` is treated as a column vector

Comparisons As a rule, comparisons on arrays are done elementwise

```
In [97]: z = np.array([2, 3])

In [98]: y = np.array([2, 3])

In [99]: z == y
Out[99]: array([ True,  True], dtype=bool)

In [100]: y[0] = 5

In [101]: z == y
Out[101]: array([False,  True], dtype=bool)

In [102]: z != y
Out[102]: array([ True, False], dtype=bool)
```

The situation is similar for >, <, >= and <=

We can also do comparisons against scalars

```
In [103]: z = np.linspace(0, 10, 5)

In [104]: z
Out[104]: array([  0. ,   2.5,   5. ,   7.5,  10. ])

In [105]: z > 3
Out[105]: array([False, False,  True,  True,  True], dtype=bool)
```

This is particularly useful for *conditional extraction*

```
In [106]: b = z > 3

In [107]: b
Out[107]: array([False, False,  True,  True,  True], dtype=bool)

In [108]: z[b]
Out[108]: array([  5. ,   7.5,  10. ])
```

Of course we can—and frequently do—perform this in one step

```
In [109]: z[z > 3]
Out[109]: array([  5. ,   7.5,  10. ])
```

Vectorized Functions NumPy provides versions of the standard functions `log`, `exp`, `sin`, etc. that act *elementwise* on arrays

```
In [110]: z = np.array([1, 2, 3])

In [111]: np.sin(z)
Out[111]: array([ 0.84147098,  0.90929743,  0.14112001])
```

This eliminates the need for explicit element-by-element loops such as

```
for i in range(n):
    y[i] = np.sin(z[i])
```

Because they act elementwise on arrays, these functions are called *vectorized functions*

In NumPy-speak, they are also called *ufuncs*, which stands for “universal functions”

As we saw above, the usual arithmetic operations (`+`, `*`, etc.) also work elementwise, and combining these with the ufuncs gives a very large set of fast elementwise functions

```
In [112]: z
Out[112]: array([1, 2, 3])

In [113]: (1 / np.sqrt(2 * np.pi)) * np.exp(- 0.5 * z**2)
Out[113]: array([ 0.24197072,  0.05399097,  0.00443185])
```

Not all user defined functions will act elementwise

For example, passing this function a NumPy array causes a `ValueError`

```
def f(x):
    return 1 if x > 0 else 0
```

In this situation you should use the vectorized NumPy function `np.where`

```
In [114]: import numpy as np

In [115]: x = np.random.randn(4)

In [116]: x
Out[116]: array([-0.25521782,  0.38285891, -0.98037787, -0.083662  ])

In [117]: np.where(x > 0, 1, 0)
Out[117]: array([0, 1, 0, 0])
```

Although it's usually better to hand code vectorized functions from vectorized NumPy operations, at a pinch you can use `np.vectorize`

```
In [118]: def f(x): return 1 if x > 0 else 0

In [119]: f = np.vectorize(f)

In [120]: f(x)                # Passing same vector x as previous example
Out[120]: array([0, 1, 0, 0])
```

NumPy Matrices

Using `np.dot` is inconvenient for expressions involving the multiplication of many matricies

For this reason, NumPy provides the `numpy.matrix` class, where the `*` operator means matrix (as opposed to elementwise) multiplication

NumPy arrays can be converted to the `numpy.matrix` class using the `np.matrix` function

```
In [122]: A = np.ones(4)

In [123]: b = np.array([0, 1])

In [124]: A.shape = (2, 2)

In [125]: b.shape = (2, 1)

In [126]: type(A)
Out[126]: numpy.ndarray

In [127]: type(b)
Out[127]: numpy.ndarray

In [128]: A = np.matrix(A)

In [129]: b = np.matrix(b)

In [130]: A * b               # Matrix multiplication
Out[130]:
matrix([[ 1.],
        [ 1.]])
```

The trick is not to get mixed up with which type (array or matrix) you are using

For this reason, the `numpy.matrix` class is often used relatively sparingly (but certainly still used)

Other NumPy Functions

NumPy provides some additional functionality related to scientific programming

For example

```
In [131]: A = np.array([[1, 2], [3, 4]])

In [132]: np.linalg.det(A)           # Compute the determinant
Out[132]: -2.0000000000000004

In [133]: np.linalg.inv(A)           # Compute the inverse
Out[133]:
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

In [134]: Z = np.random.randn(10000)  # Generate standard normals

In [135]: y = np.random.binomial(10, 0.5, size=1000)    # 1,000 draws from Bin(10, 0.5)

In [136]: y.mean()
Out[136]: 5.0369999999999999
```

However, all of this functionality is also available in SciPy, a collection of modules that build on top of NumPy

We'll cover the SciPy versions in more detail *soon*

Exercises

Exercise 1 Consider evaluating the polynomial expression

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_N x^N = \sum_{n=0}^{N} a_n x^n \tag{1.7}$$

at some given x (and with given coefficients)

Write a function which takes the vector of coefficients and a number x and returns the value $p(x)$

- This can be done using `np.poly1d`, but for the sake of the exercise don't use this class
- Avoid `for` and `while` loops to get more speed
- Hint: Use `np.cumprod()`

Solution: *View solution*

Exercise 2 Let `q` be a NumPy array of length `n` with `q.sum() == 1`

Suppose that `q` represents a probability mass function

We wish to generate a discrete random variable x such that $\mathbb{P}\{x = i\} = q_i$

In other words, `x` takes values in `range(len(q))` and `x = i` with probability `q[i]`

The standard (inverse transform) algorithm is as follows:

- Divide the unit interval $[0, 1]$ into n subintervals $I_0, I_1, \ldots, I_{n-1}$ such that the length of I_i is q_i
- Draw a uniform random variable U on $[0, 1]$ and return the i such that $U \in I_i$

The probability of drawing i is the length of I_i, which is equal to q_i

We can implement the algorithm as follows

```
from random import uniform

def sample(q):
    a = 0.0
    U = uniform(0, 1)
    for i in range(len(q)):
        if a < U <= a + q[i]:
            return i
        a = a + q[i]
```

If you can't see how this works, try thinking through the flow for a simple example, such as `q = [0.25, 0.75]` It helps to sketch the intervals on paper

Your exercise is to speed it up using NumPy, avoiding explicit loops

- Hint: Use `np.searchsorted` and `np.cumsum`

If you can, implement the functionality as a class called `discreteRV`, where

- the data for an instance of the class is the vector of probabilities `q`
- the class has a `draw()` method, which returns one draw according to the algorithm described above

If you can, write the method so that `draw(k)` returns `k` draws from `q`

Solution: *View solution*

Exercise 3 Recall our *earlier discussion* of the empirical distribution function

We came up with the implementation

```
class ecdf:

    def __init__(self, observations):
        self.observations = observations

    def __call__(self, x):
        counter = 0.0
        for obs in self.observations:
            if obs <= x:
                counter += 1
        return counter / len(self.observations)
```

Your task is to

1. Make the `__call__` method more efficient using NumPy
2. Add a method which plots the ECDF over $[a, b]$, where a and b are method parameters

Solution: *View solution*

SciPy

SciPy builds on top of NumPy to provide common tools for scientific programming, such as

- linear algebra
- numerical integration

- interpolation
- optimization
- distributions and random number generation
- signal processing
- etc., etc

Like NumPy, SciPy is stable, mature and widely used

Many SciPy routines are thin wrappers around industry-standard Fortran libraries such as LAPACK, BLAS, etc.

It’s not really necessary to “learn” SciPy as a whole—a better approach is to learn each relevant feature as required

You can browse from the top of the documentation tree to see what’s available

In this lecture we aim only to highlight some useful parts of the package

SciPy versus NumPy

SciPy is a package that contains various tools that are built on top of NumPy, using its array data type and related functionality

It’s also the case that when we import SciPy we also get NumPy, as can be seen from the SciPy initialization file

```
# Import numpy symbols to scipy name space
from numpy import *
from numpy.random import rand, randn
from numpy.fft import fft, ifft
from numpy.lib.scimath import *
# Remove the linalg imported from numpy so that the scipy.linalg package can be
# imported.
del linalg
```

In fact most of SciPy resides in its subpackages

- `scipy.optimize`, `scipy.integrate`, `scipy.stats`, etc.

We will review the major subpackages below

Note that these subpackages need to be imported separately

```
import scipy.optimize
from scipy.integrate import quad
```

Although SciPy imports NumPy, the standard approach is to start scientific programs with

```
import numpy as np
```

and then import bits and pieces from SciPy as needed

```
from scipy.integrate import quad
from scipy.optimize import brentq
# etc
```

This approach helps clarify what functionality belongs to what package, and we will follow it in these lectures

Statistics

The scipy.stats subpackage supplies

- numerous random variable objects (densities, cumulative distributions, random sampling, etc.)
- some estimation procedures
- some statistical tests

Random Variables and Distributions Recall that numpy.random provides functions for generating random variables

```
In [1]: import numpy as np

In [2]: np.random.beta(5, 5, size=3)
Out[2]: array([ 0.6167565 ,  0.67994589,  0.32346476])
```

This generates a draw from the distribution below when a, b = 5, 5

$$f(x; a, b) = \frac{x^{(a-1)}(1-x)^{(b-1)}}{\int_0^1 u^{(a-1)} u^{(b-1)} du} \qquad (0 \leq x \leq 1) \tag{1.8}$$

Sometimes we need access to the density itself, or the cdf, the quantiles, etc.

For this we can use scipy.stats, which provides all of this functionality as well as random number generation in a single consistent interface

Here's an example of usage

```
import numpy as np
from scipy.stats import beta
from matplotlib.pyplot import hist, plot, show
q = beta(5, 5)      # Beta(a, b), with a = b = 5
obs = q.rvs(2000)   # 2000 observations
hist(obs, bins=40, normed=True)
grid = np.linspace(0.01, 0.99, 100)
plot(grid, q.pdf(grid), 'k-', linewidth=2)
show()
```

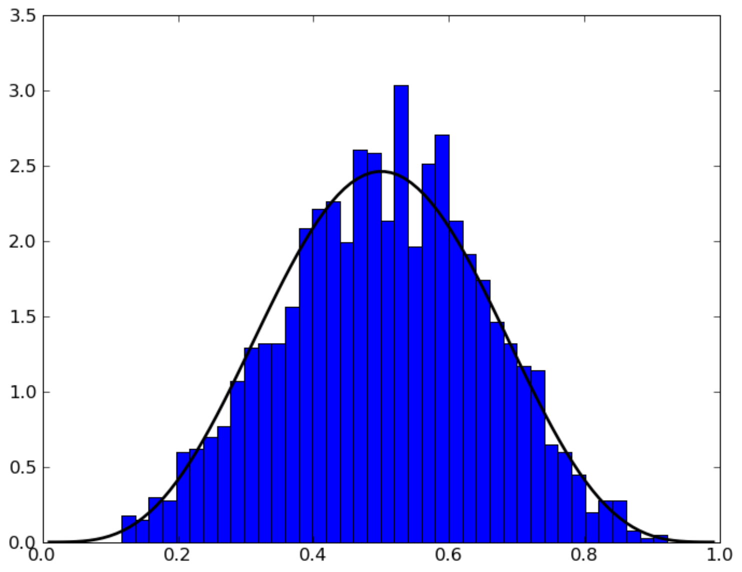
The following plot is produced

In this code we created a so-called rv_frozen object, via the call q = beta(5, 5)

The "frozen" part of the notation related to the fact that q represents a particular distribution with a particular set of parameters

Once we've done so, we can then generate random numbers, evaluate the density, etc., all from this fixed distribution

```
In [14]: q.cdf(0.4)       # Cumulative distribution function
Out[14]: 0.2665676800000002

In [15]: q.pdf(0.4)       # Density function
Out[15]: 2.0901888000000004

In [16]: q.ppf(0.8)       # Quantile (inverse cdf) function
Out[16]: 0.63391348346427079

In [17]: q.mean()
Out[17]: 0.5
```



The general syntax for creating these objects is

```
identifier = scipy.stats.distribution_name(shape_parameters)
```

where `distribution_name` is one of the distribution names in scipy.stats

There are also two keyword arguments, `loc` and `scale`:

```
identifier = scipy.stats.distribution_name(shape_parameters, `loc=c`, `scale=d`)
```

These transform the original random variable X into $Y = c + dX$

The methods `rvs`, `pdf`, `cdf`, etc. are transformed accordingly

Before finishing this section, we note that there is an alternative way of calling the methods described above

For example, the previous code can be replaced by

```
import numpy as np
from scipy.stats import beta
from matplotlib.pyplot import hist, plot, show
obs = beta.rvs(5, 5, size=2000)    # 2000 observations
hist(obs, bins=40, normed=True)
grid = np.linspace(0.01, 0.99, 100)
plot(grid, beta.pdf(grid, 5, 5), 'k-', linewidth=2)
show()
```

Other Goodies in scipy.stats There are also many statistical functions in `scipy.stats`

For example, `scipy.stats.linregress` implements simple linear regression

```
In [19]: from scipy.stats import linregress

In [20]: x = np.random.randn(200)

In [21]: y = 2 * x + 0.1 * np.random.randn(200)
In [22]: gradient, intercept, r_value, p_value, std_err = linregress(x, y)

In [23]: gradient, intercept
Out[23]: (1.9962554379482236, 0.008172822032671799)
```

To see the full list of statistical functions, consult the documentation

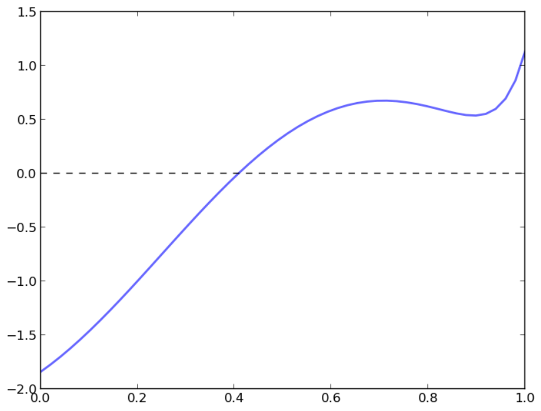
Roots and Fixed Points

A *root* of a real function f on $[a, b]$ is an $x \in [a, b]$ such that $f(x) = 0$

For example, if we plot the function

$$f(x) = \sin(4(x - 1/4)) + x + x^{20} - 1 \tag{1.9}$$

with $x \in [0, 1]$ we get



The unique root is approximately 0.408

Let's consider some numerical techniques for finding roots

Bisection One of the most common algorithms for numerical root finding is *bisection*

To understand the idea, recall the well known game where

- Player A thinks of a secret number between 1 and 100
- Player B asks if it's less than 50
  - If yes, B asks if it's less than 25
  - If no, B asks if it's less than 75

And so on

This is bisection. Here's a fairly simplistic implementation of the algorithm in Python

```
def bisect(f, a, b, tol=10e-5):
    """
    Implements the bisection root finding algorithm, assuming that f is a
    real-valued function on [a, b] satisfying f(a) < 0 < f(b).
    """
    lower, upper = a, b
    while upper - lower > tol:
        middle = 0.5 * (upper + lower)
        if f(middle) > 0:  # Implies root is between lower and middle
            lower, upper = lower, middle
        else:              # Implies root is between middle and upper
            lower, upper = middle, upper
    return 0.5 * (upper + lower)
```

In fact SciPy provides it's own bisection function, which we now test using the function f defined in (1.9)

```
In [24]: from scipy.optimize import bisect

In [25]: f = lambda x: np.sin(4 * (x - 0.25)) + x + x**20 - 1

In [26]: bisect(f, 0, 1)
Out[26]: 0.40829350427936706
```

The Newton-Raphson Method Another very common root-finding algorithm is the Newton-Raphson method

In SciPy this algorithm is implemented by `scipy.newton`

Unlike bisection, the Newton-Raphson method uses local slope information

This is a double-edged sword:

- When the function is well-behaved, the Newton-Raphson method is faster than bisection
- When the function is less well-behaved, the Newton-Raphson might fail

Let's investigate this using the same function f, first looking at potential instability

```
In [27]: from scipy.optimize import newton

In [28]: newton(f, 0.2)   # Start the search at initial condition x = 0.2
Out[28]: 0.40829350427935679

In [29]: newton(f, 0.7)   # Start the search at x = 0.7 instead
Out[29]: 0.70017000000002816
```

The second initial condition leads to failure of convergence

On the other hand, using IPython's `timeit` magic, we see that `newton` can be much faster

```
In [32]: timeit bisect(f, 0, 1)
1000 loops, best of 3: 261 us per loop

In [33]: timeit newton(f, 0.2)
10000 loops, best of 3: 60.2 us per loop
```

Hybrid Methods So far we have seen that the Newton-Raphson method is fast but not robust

This bisection algorithm is robust but relatively slow

This illustrates a general principle

- If you have specific knowledge about your function, you might be able to exploit it to generate efficiency
- If not, then algorithm choice involves a trade-off between speed of convergence and robustness

In practice, most default algorithms for root finding, optimization and fixed points use *hybrid* methods

These methods typically combine a fast method with a robust method in the following manner:

1. Attempt to use a fast method
2. Check diagnostics
3. If diagnostics are bad, then switch to a more robust algorithm

In `scipy.optimize`, the function `brentq` is such a hybrid method, and a good default

```
In [35]: brentq(f, 0, 1)
Out[35]: 0.40829350427936706

In [36]: timeit brentq(f, 0, 1)
10000 loops, best of 3: 63.2 us per loop
```

Here the correct solution is found and the speed is almost the same as `newton`

Multivariate Root Finding Use `scipy.optimize.fsolve`, a wrapper for a hybrid method in MINPACK

See the documentation for details

Fixed Points SciPy has a function for finding (scalar) fixed points too

```
In [1]: from scipy.optimize import fixed_point

In [2]: fixed_point(lambda x: x**2, 10.0)  # 10.0 is an initial guess
Out[2]: 1.0
```

If you don't get good results, you can always switch back to the `brentq` root finder, since the fixed point of a function f is the root of $g(x) := x - f(x)$

Optimization

Most numerical packages provide only functions for *minimization*

Maximization can be performed by recalling that the maximizer of a function f on domain D is the minimizer of $-f$ on D

Minimization is closely related to root finding: For smooth functions, interior optima correspond to roots of the first derivative

The speed/robustness trade-off described above is present with numerical optimization too

Unless you have some prior information you can exploit, it's usually best to use hybrid methods

For constrained, univariate (i.e., scalar) minimization, a good hybrid option is `fminbound`

```
In [9]: from scipy.optimize import fminbound

In [10]: fminbound(lambda x: x**2, -1, 2)  # Search in [-1, 2]
Out[10]: 0.0
```

Multivariate Optimization Multivariate local optimizers include `minimize`, `fmin`, `fmin_powell`, `fmin_cg`, `fmin_bfgs`, and `fmin_ncg`

Constrained multivariate local optimizers include `fmin_l_bfgs_b`, `fmin_tnc`, `fmin_cobyla`

See the documentation for details

Integration

Most numerical integration methods work by compute the integral of an approximating polynomial

The resulting error depends how well the polynomial fits integrand, which in turn depends on how "regular" the integrand is

In SciPy, the relevant module for numerical integration is `scipy.integrate`

A good default for univariate integration is `quad`

```
In [13]: from scipy.integrate import quad

In [14]: integral, error = quad(lambda x: x**2, 0, 1)

In [15]: integral
Out[15]: 0.33333333333333337
```

In fact `quad` is an interface to a very standard numerical integration routine in the Fortran library QUADPACK

It uses Clenshaw-Curtis quadrature, based on expansion in terms of Chebychev polynomials

There are other options for univeriate integration—a useful one is `fixed_quad`, which is fast and hence works well inside `for` loops

There are also functions for multivariate integration

See the documentation for more details

Linear Algebra

We saw that NumPy provides a module for linear algebra called `linalg`

SciPy also provides a module for linear algebra with the same name

The latter is not an exact superset of the former, but overall it has more functionality

We leave you to investigate the set of available routines

Exercises

Exercise 1 Recall that we previously discussed the concept of *recusive function calls*

Write a recursive implementation of the bisection function described above, which we repeat here for convenience

```
def bisect(f, a, b, tol=10e-5):
    """
    Implements the bisection root finding algorithm, assuming that f is a
    real-valued function on [a, b] satisfying f(a) < 0 < f(b).
    """
    lower, upper = a, b
    while upper - lower > tol:
        middle = 0.5 * (upper + lower)
        if f(middle) > 0:  # Implies root is between lower and middle
            lower, upper = lower, middle
        else:              # Implies root is between middle and upper
            lower, upper = middle, upper
    return 0.5 * (upper + lower)
```

Test it on the function `f = lambda x: np.sin(4 * (x - 0.25)) + x + x**20 - 1` discussed above

Solution: *View solution*

Matplotlib

Overview

We've already generated quite a few figures in these lectures using Matplotlib

Matplotlib is interesting in that it provides two fairly different interfaces

The first is designed to mimic MATLAB graphics functionality

The second is object oriented, and more Pythonic

In this lecture we'll cover both, with a focus on the second method

The MATLAB-style API

Matplotlib is very easy to get started with, thanks to its simple MATLAB-style API

Here's an example plot

```
from pylab import *
x = linspace(0, 10, 200)
y = sin(x)
plot(x, y, 'b-', linewidth=2)
show()
```

The figure it generates looks as follows

If you've run these commands inside the IPython notebook with the `--pylab inline` flag, the figure will appear embedded in your browser

If you've run these in IPython without this flag, it will appear as a separate window

The buttons at the bottom of the window allow you to manipulate the figure and then save it if you wish

Comments:

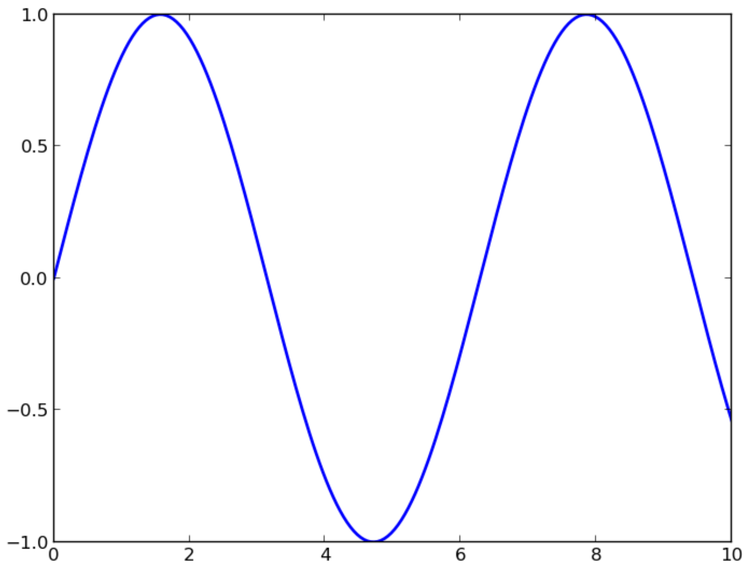
- The `pylab` module combines core parts of `matplotlib`, `numpy` and `scipy`
- Hence `from pylab import *` pulls NumPy functions like `linspace` into the global namespace

Almost everyone starts working with Matplotlib using this MATLAB style

The Object-Oriented Approach

The MATLAB style API is simple and convenient, but can be a bit limited

It is also slightly un-Pythonic


1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10

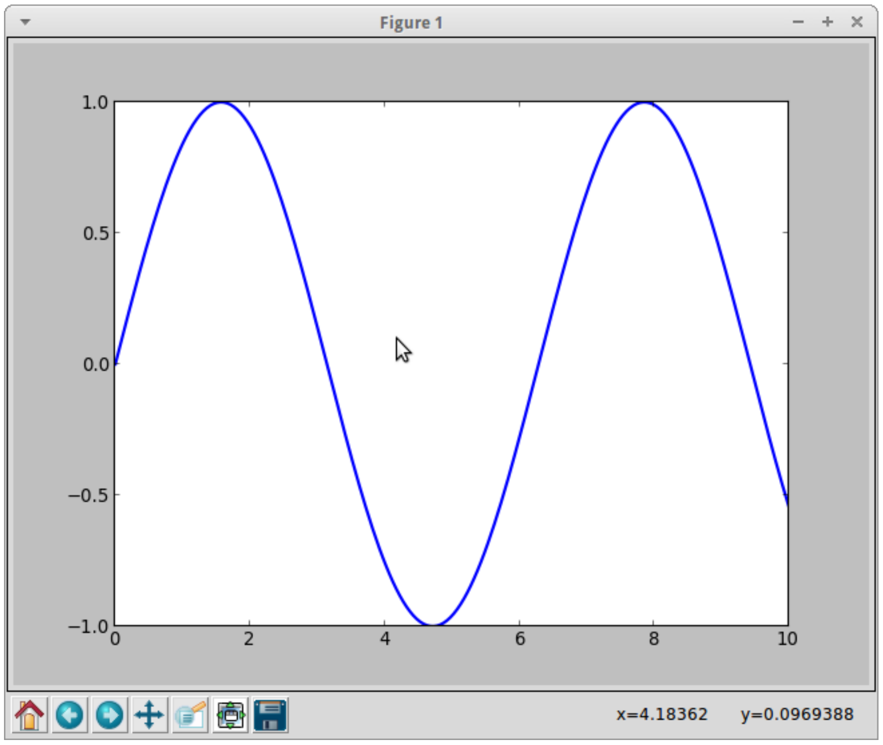


Figure 1
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10
x=4.18362
y=0.0969388

- We are pulling lots of names into the global namespace
- There are a lot of implicit calls behind the scenes—Python favors explicit over implicit

Hence it's worthwhile adopting the alternative, object oriented API

Here's the code corresponding to the preceding figure using this second approach

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'b-', linewidth=2)
fig.show()
```

You can see there's a bit more typing, but the more explicit declarations will help when we need fine-grained control

Details:

- the form of the import statement `import matplotlib.pyplot as plt` is standard
- Here the call `fig, ax = plt.subplots()` returns a pair, where
  - `fig` is a `Figure` instance—like a blank canvas
  - `ax` is a `AxesSubplot` instance—think of a frame for plotting in
- The `plot()` function is actually a method of `ax`, while `show()` is a method of fig

Variations Here we've changed the line to red and added a legend

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', linewidth=2, label='sine function', alpha=0.6)
ax.legend()
fig.show()
```

We've also used `alpha` to make the line slightly transparent—which makes it look smoother

Unfortunately the legend is obscuring the line

This can be fixed by replacing `ax.legend()` with `ax.legend(loc='upper center')`

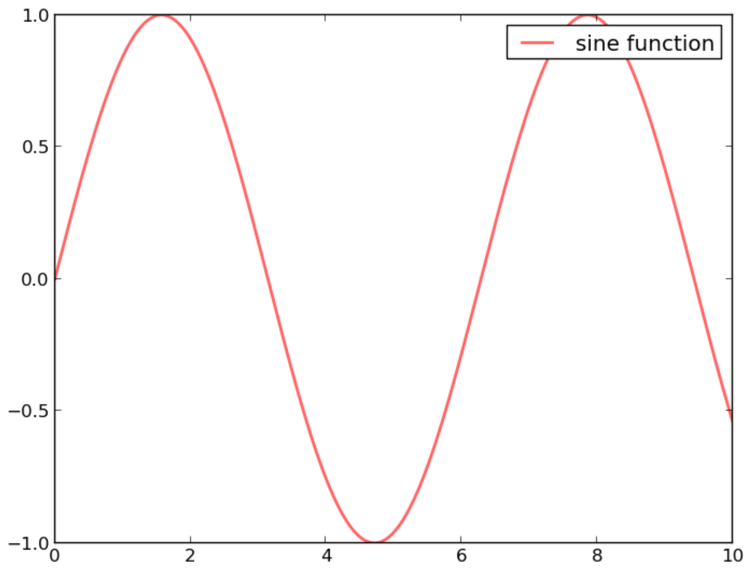
If everthing is properly configured, then adding LaTeX is trivial

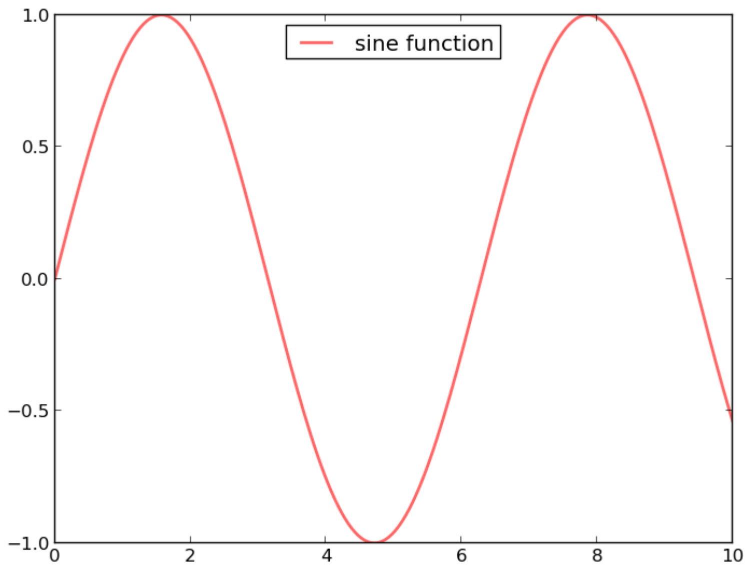
```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', linewidth=2, label=r'$y=\sin(x)$', alpha=0.6)
ax.legend(loc='upper center')
fig.show()
```

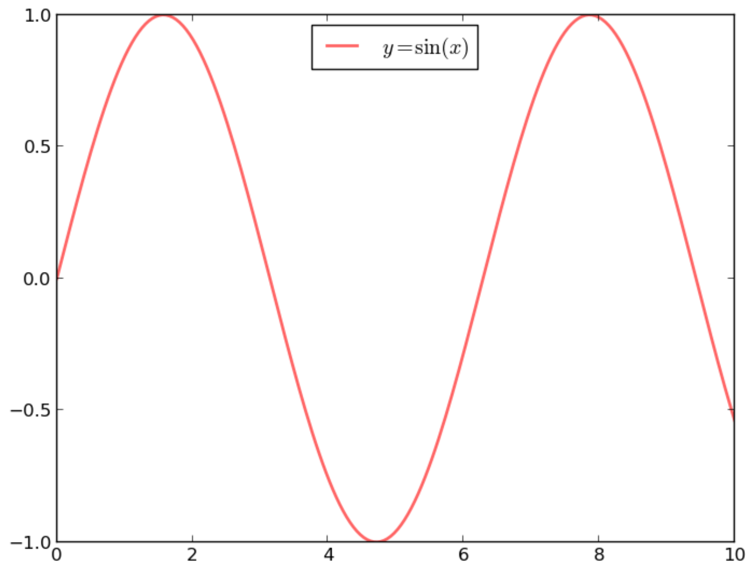
The `r` in front of the label string tells Python that this is a raw string

The figure now looks as follows

Controlling the ticks, adding titles and so on is also straightforward


sine function
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10


sine function
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10



```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', linewidth=2, label=r'$y=\sin(x)$', alpha=0.6)
ax.legend(loc='upper center')
ax.set_yticks([-1, 0, 1])
ax.set_title('Test plot')
fig.show()
```

It's straightforward to generate mutiple plots on the same axes

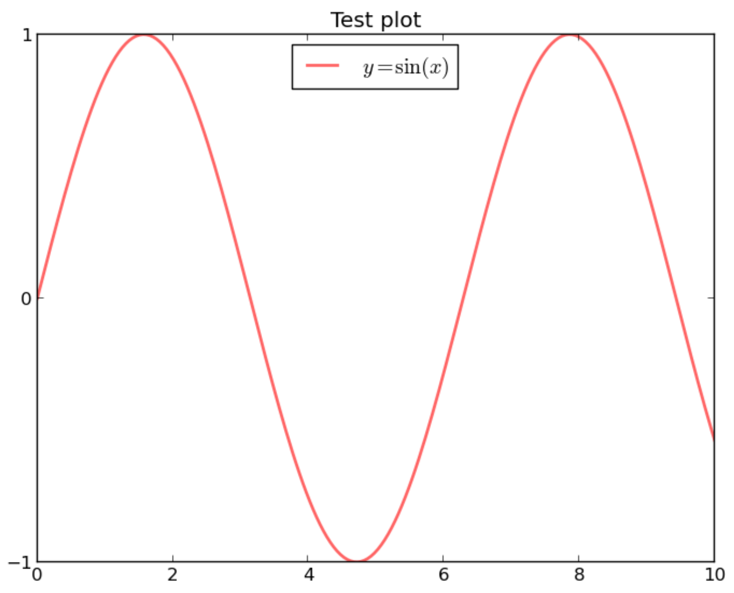
Here's an example that randomly generates three normal densities and adds a label with their mean

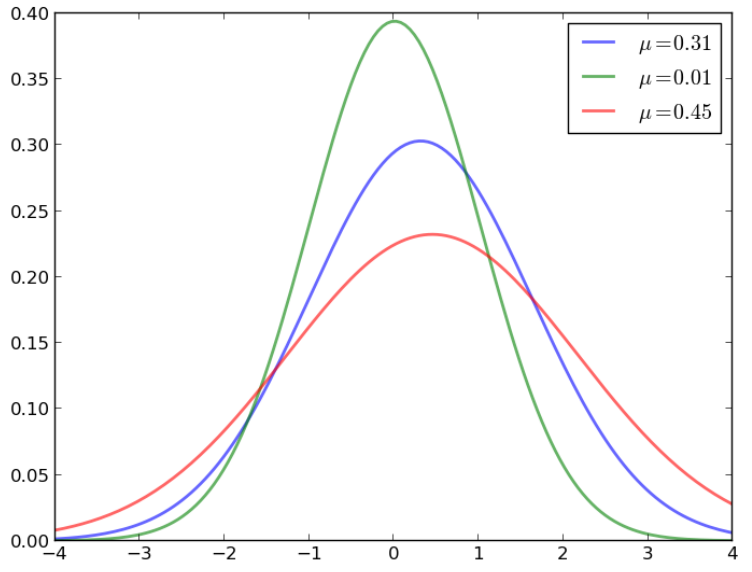
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm
from random import uniform
fig, ax = plt.subplots()
x = np.linspace(-4, 4, 150)
for i in range(3):
    m, s = uniform(-1, 1), uniform(1, 2)
    y = norm.pdf(x, loc=m, scale=s)
    current_label = r'$\mu = {0:.2f}$'.format(m)
    ax.plot(x, y, linewidth=2, alpha=0.6, label=current_label)
ax.legend()
fig.show()
```

At other times we want multiple subplots on the one figure

Here's an example that generates 6 histograms

Notice the lines at the start used to control the LaTeX font


Test plot
$y = \sin(x)$
1
0
−1
0
2
4
6
8
10


$\mu = 0.31$
$\mu = 0.01$
$\mu = 0.45$
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−4
−3
−2
−1
0
1
2
3
4

```
from matplotlib import rc
rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm
from random import uniform
num_rows, num_cols = 3, 2
fig, axes = plt.subplots(num_rows, num_cols, figsize=(8, 12))
for i in range(num_rows):
    for j in range(num_cols):
        m, s = uniform(-1, 1), uniform(1, 2)
        x = norm.rvs(loc=m, scale=s, size=100)
        axes[i, j].hist(x, alpha=0.6, bins=20)
        t = r'$\mu = {0:.1f}, \quad \sigma = {1:.1f}$'.format(m, s)
        axes[i, j].set_title(t)
        axes[i, j].set_xticks([-4, 0, 4])
        axes[i, j].set_yticks([])
fig.show()
```

The output looks as follows

Visit the Matplotlib gallery to see more examples

Pandas

Overview of This Lecture

Pandas is a package of fast, efficient data analysis tools for Python

Just as NumPy provides the basic array type plus core array operations, pandas defines some fundamental structures for working with data and endows them with methods that form the first steps of data analysis

The most important data type defined by pandas is a `DataFrame`, which is an object for storing related columns of data

In this sense, you can think of a `DataFrame` as analogous to a (highly optimized) Excel spreadsheet, or as a structure for storing the $\mathbf{X}$ matrix in a linear regression

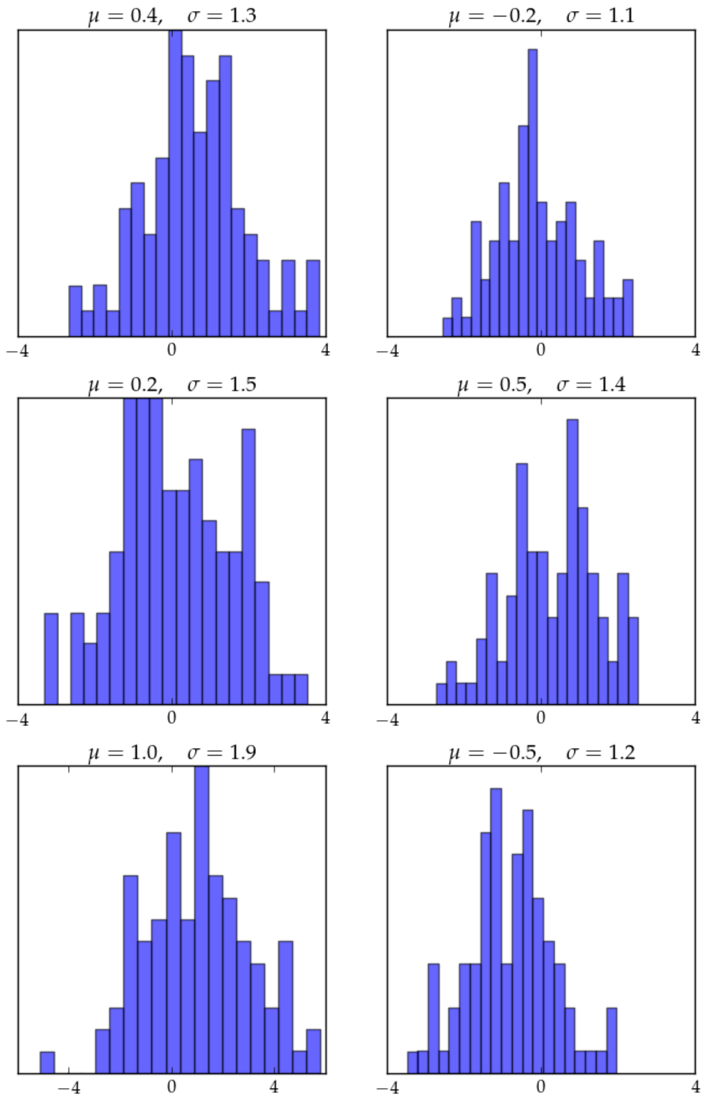
In the same way that NumPy specializes in basic array operations and leaves the rest of scientific tool development to other packages (e.g., SciPy, Matplotlib), pandas focuses on the fundamental data types and their methods, leaving other packages to add more sophisticated statistical functionality

The strengths of pandas lie in

- reading in data
- manipulating rows and columns
- adjusting indices
- working with dates and time series
- sorting, grouping, re-ordering and general data munging
- dealing with missing values, etc., etc.

This lecture will provide a basic introduction

Throughout the lecture we will assume that the following imports have taken place


$\mu = 0.4, \quad \sigma = 1.3$
$\mu = -0.2, \quad \sigma = 1.1$
$\mu = 0.2, \quad \sigma = 1.5$
$\mu = 0.5, \quad \sigma = 1.4$
$\mu = 1.0, \quad \sigma = 1.9$
$\mu = -0.5, \quad \sigma = 1.2$
−4
0
4

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

Series

Perhaps the two most important data types defined by pandas are the `DataFrame` and `Series` types

You can think of a `Series` as a "column" of data, such as a collection of observations on a single variable

```
In [4]: s = pd.Series(np.random.randn(4), name='daily returns')

In [5]: s
Out[5]:
0    0.430271
1    0.617328
2   -0.265421
3   -0.836113
Name: daily returns
```

Here you can imagine the indices `0, 1, 2, 3` as indexing four listed companies, and the values being daily returns on their shares

Pandas `Series` are built on top of NumPy arrays, and support many similar operations

```
In [6]: s * 100
Out[6]:
0    43.027108
1    61.732829
2   -26.542104
3   -83.611339
Name: daily returns

In [7]: np.abs(s)
Out[7]:
0    0.430271
1    0.617328
2    0.265421
3    0.836113
Name: daily returns
```

But `Series` provide more than NumPy arrays

Not only do they have some additional (statistically orientated) methods

```
In [8]: s.describe()
Out[8]:
count    4.000000
mean    -0.013484
std      0.667092
min     -0.836113
25%     -0.408094
50%      0.082425
75%      0.477035
max      0.617328
```

But their indices are more flexible

```
In [9]: s.index = ['AMZN', 'AAPL', 'MSFT', 'GOOG']

In [10]: s
Out[10]:
AMZN    0.430271
AAPL    0.617328
MSFT   -0.265421
GOOG   -0.836113
Name: daily returns
```

Viewed in this way, `Series` are like fast, efficient Python dictionaries (with the restriction that the items in the dictionary all have the same type—in this case, floats)

In fact you can use much of the same syntax as Python dictionaries

```
In [11]: s['AMZN']
Out[11]: 0.43027108469945924

In [12]: s['AMZN'] = 0

In [13]: s
Out[13]:
AMZN    0.000000
AAPL    0.617328
MSFT   -0.265421
GOOG   -0.836113
Name: daily returns

In [14]: 'AAPL' in s
Out[14]: True
```

DataFrames

As mentioned above a `DataFrame` is somewhat like a spreadsheet, or a structure for storing the data matrix $\mathbf{X}$ in a regression

While a `Series` is one individual column of data, a `DataFrame` is all the columns

Let's look at an example, reading in data from the CSV file `test_pwd.csv` in the main repository

Here's the contents of `test_pwd.csv`, which is a small except from the Penn World Tables

```
"country","country isocode","year","POP","XRAT","tcgdp","cc","cg"
"Argentina","ARG","2000","37335.653","0.9995","295072.21869","75.716805379","5.5788042896"
"Australia","AUS","2000","19053.186","1.72483","541804.6521","67.759025993","6.7200975332"
"India","IND","2000","1006300.297","44.9416","1728144.3748","64.575551328","14.072205773"
"Israel","ISR","2000","6114.57","4.07733","129253.89423","64.436450847","10.266688415"
"Malawi","MWI","2000","11801.505","59.543808333","5026.2217836","74.707624181","11.658954494"
"South Africa","ZAF","2000","45064.098","6.93983","227242.36949","72.718710427","5.7265463933"
"United States","USA","2000","282171.957","1","9898700","72.347054303","6.0324539789"
"Uruguay","URY","2000","3219.793","12.099591667","25255.961693","78.978740282","5.108067988"
```

Here we're in IPython, so we have access to shell commands such as `ls`, as well as the usual Python commands

```
In [15]: ls test_pw*  # List all files starting with 'test_pw' -- check CSV file is in present worki
test_pwt.csv
```

Now let's read the data in using pandas' `read_csv` function

```
In [28]: df = pd.read_csv('test_pwt.csv')

In [29]: type(df)
Out[29]: pandas.core.frame.DataFrame

In [30]: df
Out[30]:
         country country isocode  year           POP       XRAT            tcgdp  cc         cg
0      Argentina             ARG  2000     37335.653   0.999500    295072.218690   0  75.716805   5.578
1      Australia             AUS  2000     19053.186   1.724830    541804.652100   1  67.759026   6.720
2          India             IND  2000   1006300.297  44.941600   1728144.374800   2  64.575551  14.072
3         Israel             ISR  2000      6114.570   4.077330    129253.894230   3  64.436451  10.266
4         Malawi             MWI  2000     11801.505  59.543808      5026.221784   4  74.707624  11.658
5   South Africa             ZAF  2000     45064.098   6.939830    227242.369490   5  72.718710   5.726
6  United States             USA  2000    282171.957   1.000000   9898700.000000   6  72.347054   6.032
7        Uruguay             URY  2000      3219.793  12.099592     25255.961693   7  78.978740   5.108
```

Let's imagine that we're only interested in population and total GDP (`tcgdp`)

One way to strip the data frame `df` down to only these variables is as follows

```
In [31]: keep = ['country', 'POP', 'tcgdp']

In [32]: df = df[keep]

In [33]: df
Out[33]:
         country          POP            tcgdp
0      Argentina    37335.653    295072.218690
1      Australia    19053.186    541804.652100
2          India  1006300.297   1728144.374800
3         Israel     6114.570    129253.894230
4         Malawi    11801.505      5026.221784
5   South Africa    45064.098    227242.369490
6  United States   282171.957   9898700.000000
7        Uruguay     3219.793     25255.961693
```

Here the index `0, 1,..., 7` is redundant, because we can use the country names as an index

To do this, first let's pull out the `country` column using the `pop` method

```
In [34]: countries = df.pop('country')

In [35]: type(countries)
Out[35]: pandas.core.series.Series

In [36]: countries
Out[36]:
0        Argentina
1        Australia
2            India
3           Israel
4           Malawi
5     South Africa
6    United States
7          Uruguay
Name: country

In [37]: df
Out[37]:
```

```
           POP           tcgdp
0    37335.653   295072.218690
1    19053.186   541804.652100
2  1006300.297  1728144.374800
3     6114.570   129253.894230
4    11801.505     5026.221784
5    45064.098   227242.369490
6   282171.957  9898700.000000
7     3219.793    25255.961693

In [38]: df.index = countries

In [39]: df
Out[39]:
                       POP           tcgdp
country
Argentina        37335.653   295072.218690
Australia        19053.186   541804.652100
India          1006300.297  1728144.374800
Israel            6114.570   129253.894230
Malawi           11801.505     5026.221784
South Africa     45064.098   227242.369490
United States   282171.957  9898700.000000
Uruguay           3219.793    25255.961693
```

Let’s give the columns slightly better names

```
n [40]: df.columns = 'population', 'total GDP'

In [41]: df
Out[41]:
                population       total GDP
country
Argentina        37335.653   295072.218690
Australia        19053.186   541804.652100
India          1006300.297  1728144.374800
Israel            6114.570   129253.894230
Malawi           11801.505     5026.221784
South Africa     45064.098   227242.369490
United States   282171.957  9898700.000000
Uruguay           3219.793    25255.961693
```

Population is in thousands, let’s revert to single units

```
In [66]: df['population'] = df['population'] * 1e3

In [67]: df
Out[67]:
                population       total GDP
country
Argentina         37335653   295072.218690
Australia         19053186   541804.652100
India           1006300297  1728144.374800
Israel             6114570   129253.894230
Malawi            11801505     5026.221784
South Africa      45064098   227242.369490
United States    282171957  9898700.000000
Uruguay            3219793    25255.961693
```

Next we're going to add a column showing real GDP per capita, multiplying by 1,000,000 as we go because total GDP is in millions

```
In [74]: df['GDP percap'] = df['total GDP'] * 1e6 / df['population']
```

```
In [75]: df
Out[75]:
                population       total GDP    GDP percap
country
Argentina         37335653   295072.218690   7903.229085
Australia         19053186   541804.652100  28436.433261
India           1006300297  1728144.374800   1717.324719
Israel             6114570   129253.894230  21138.672749
Malawi            11801505     5026.221784    425.896679
South Africa      45064098   227242.369490   5042.647686
United States    282171957  9898700.000000  35080.381854
Uruguay            3219793    25255.961693   7843.970620
```

One of the nice things about pandas `DataFrame` and `Series` objects is that they have methods for plotting and visualization that work through Matplotlib

Let's now generate a bar plot of GDP per capita

```
In [76]: df['GDP percap'].plot(kind='bar')
Out[76]: <matplotlib.axes.AxesSubplot at 0x2f22ed0>
```

The next two lines might be unnecessary if you're using IPython notebook

```
In [77]: import matplotlib.pyplot as plt
```

```
In [78]: plt.show()
```

The following figure is produced

At the moment the data frame is ordered alphabetically on the countries—let's change it to GDP per capita

```
In [83]: df = df.sort_index(by='GDP percap', ascending=False)
```

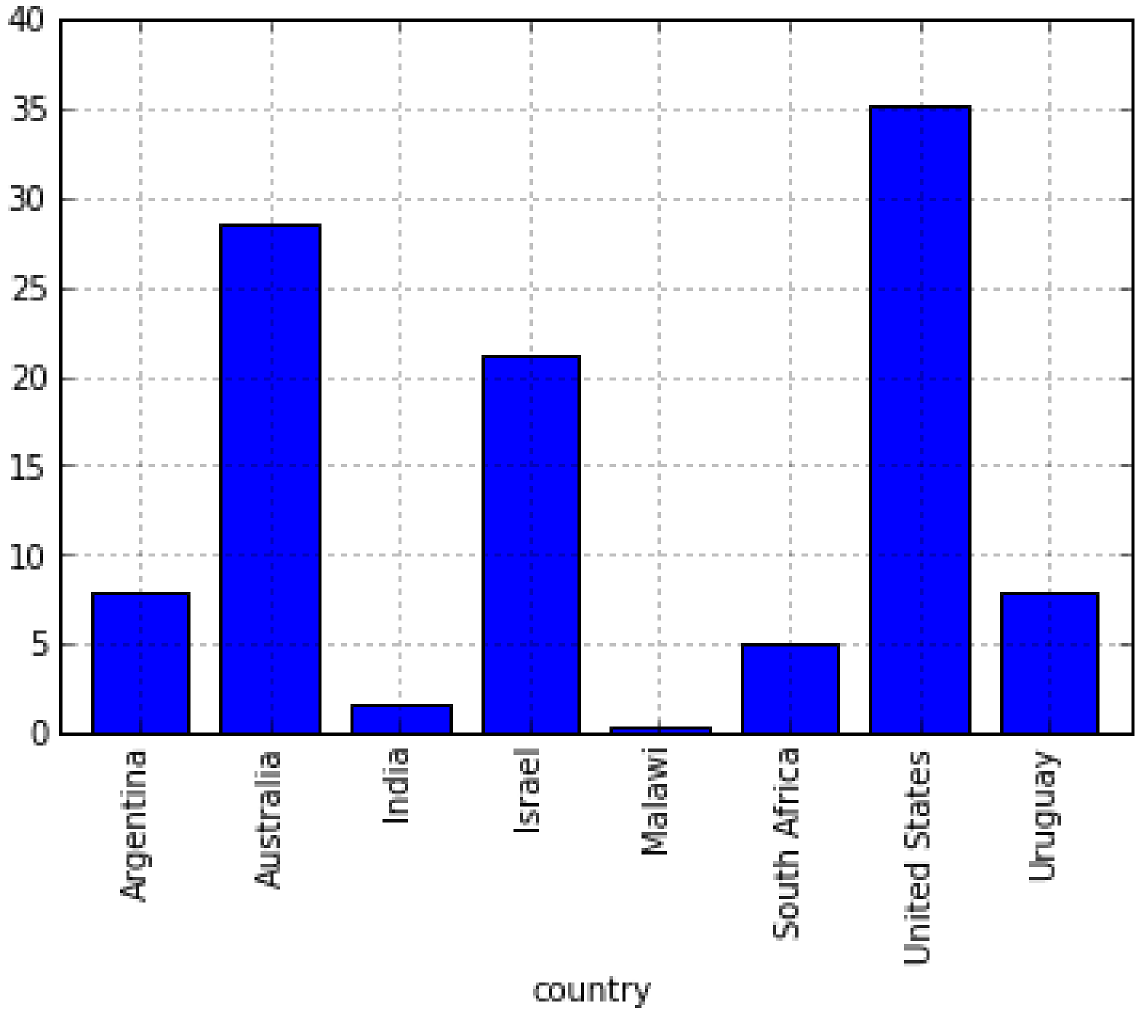
```
In [84]: df
Out[84]:
                population       total GDP    GDP percap
country
United States    282171957  9898700.000000  35080.381854
Australia         19053186   541804.652100  28436.433261
Israel             6114570   129253.894230  21138.672749
Argentina         37335653   295072.218690   7903.229085
Uruguay            3219793    25255.961693   7843.970620
South Africa      45064098   227242.369490   5042.647686
India           1006300297  1728144.374800   1717.324719
Malawi            11801505     5026.221784    425.896679
```

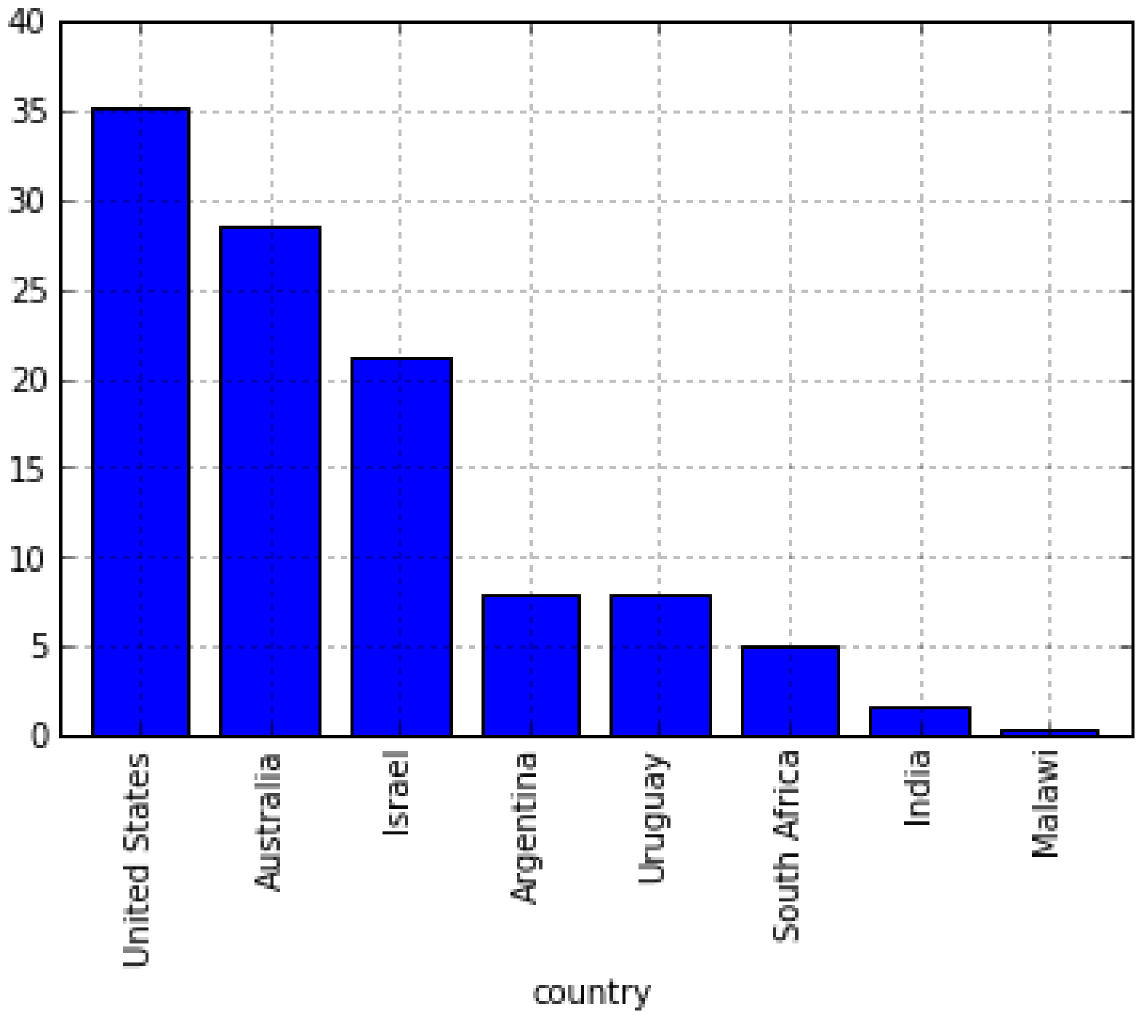
Plotting as before now yields

We'll leave our discussion of pandas for the moment and come back to it later on

1.4 Part 3: Introductory Applications

This section of the course contains relatively simple applications, one purpose of which is to teach you more about the Python programming environment


40
35
30
25
20
15
10
5
0
Argentina
Australia
India
Israel
Malawi
South Africa
United States
Uruguay
country


40
35
30
25
20
15
10
5
0
United States
Australia
Israel
Argentina
Uruguay
South Africa
India
Malawi
country

Further details to be written

1.4.1 Lectures

Shortest Paths and Dynamic Programming

Note: To be edited. Mention that one purpose of the lecture is to provide practice working with dictionaries.

Let's consider the problem of how to find the shortest path on a graph

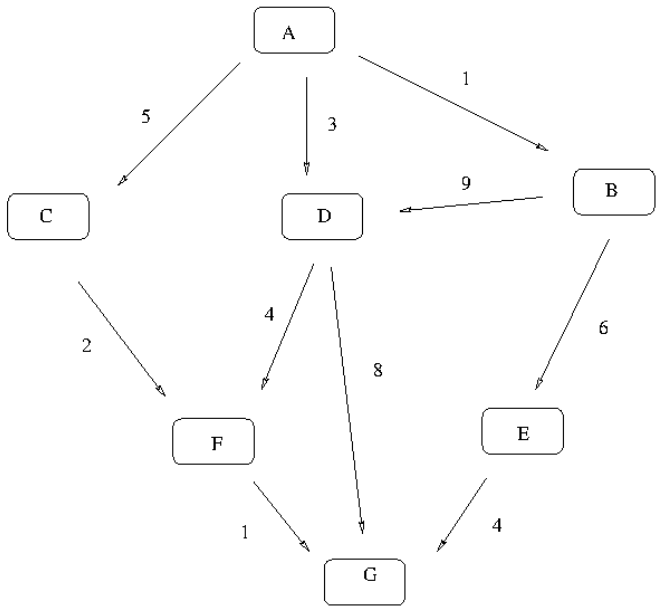
This is a well-known problem with applications in

- Finding directions between locations
- Operations research, robotics, etc.

For us it provides a simple introduction to the logic of dynamic programming, which is one of our key topics

Outline of the Problem

Consider the following graph



We wish to travel from node (vertex) A to node G at minimum cost

- Arrows (edges) indicate the movements we can take
- Numbers next to edges indicate the cost of traveling that edge

Possible interpretations of the graph include

- Minimum cost for supplier to reach a destination
- Routing of packets on the internet (minimize time)
- Etc., etc.

For this simple graph, a quick scan of the edges shows that the optimal paths are

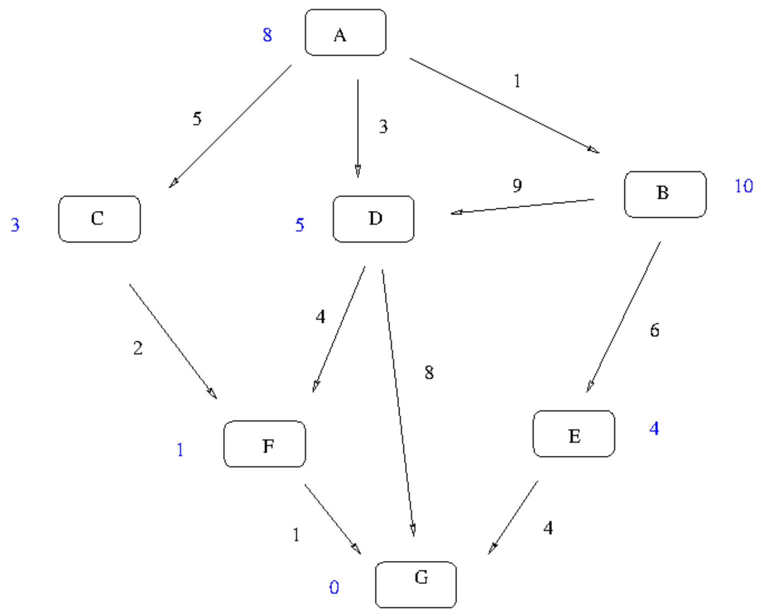
- A, C, F, G at cost 8
- A, D, F, G at cost 8

Finding Least-Cost Paths

For large graphs we need a systematic solution

Let $J(v)$ denote the minimum cost-to-go from node v, understood as the total cost from v if we take the best route

Suppose that we know $J(v)$ for each node v, as shown below for the graph from the preceding example



Note that $J(G) = 0$

Intuitively, the best path can now be found as follows

- Start at A
- From node v, move to any node that solves

$$\min_{w \in F_v} \{c(v, w) + J(w)\} \tag{1.10}$$

where

- F_v is the set of nodes which can be reached from v in one step
- $c(v, w)$ is the cost of traveling from v to w

Hence, if we know the function J, then finding the best path is almost trivial

But how to find J?

Some thought will convince you that, for every node v, the function J satisfies

$$J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\} \tag{1.11}$$

This is known as Bellman's equation

- That is, J is the solution to Bellman's equation
- There are algorithms for finding this solution

Solving for J

The standard algorithm for finding J is to start with

$$J_0(v) = M \text{ if } v \neq \text{ destination, else } J_0(v) = 0 \tag{1.12}$$

where M is some large number

Now we use the following algorithm

1. Set $n = 0$
2. Set $J_{n+1}(v) = \min_{w \in F_v} \{c(v, w) + J_n(w)\}$ for all v
3. If J_{n+1} and J_n are not equal then increment n, go to 2

In general, this sequence converges to J—the proof is omitted

Exercises Use this algorithm to find the optimal path (and its cost) for `this graph`

Here the line `node0, node1 0.04, node8 11.11, node14 72.21` means that from node0 we can go to

- node1 at cost 0.04
- node8 at cost 11.11
- node14 at cost 72.21

And so on, and so on

According to our calucations, the optimal path and its cost are like `this`

Your code should replicate this result

Solution

```
"""
Source: QEwP by John Stachurski and Thomas J. Sargent
Date: April 2013
File: solution_shortpath.py
"""
```

```python
def read_graph():
    """ Read in the graph from the data file.  The graph is stored
    as a dictionary, where the keys are the nodes, and the values
    are a list of pairs (d, c), where d is a node and c is a number.
    If (d, c) is in the list for node n, then d can be reached from
    n at cost c.
    """
    graph = {}
    infile = open('graph.txt')
    for line in infile:
        elements = line.split(',')
        node = elements.pop(0).strip()
        graph[node] = []
        if node != 'node99':
            for element in elements:
                destination, cost = element.split()
                graph[node].append((destination.strip(), float(cost)))
    infile.close()
    return graph

def update_J(J, graph):
    "The Bellman operator."
    next_J = {}
    for node in graph:
        if node == 'node99':
            next_J[node] = 0
        else:
            next_J[node] = min(cost + J[dest] for dest, cost in graph[node])
    return next_J

def print_best_path(J, graph):
    """ Given a cost-to-go function, computes the best path.  At each node n,
    the function prints the current location, looks at all nodes that can be
    reached from n, and moves to the node m which minimizes c + J[m], where c
    is the cost of moving to m.
    """
    sum_costs = 0
    current_location = 'node0'
    while current_location != 'node99':
        print current_location
        running_min = 1e100  # Any big number
        for destination, cost in graph[current_location]:
            cost_of_path = cost + J[destination]
            if cost_of_path < running_min:
                running_min = cost_of_path
                minimizer_cost = cost
                minimizer_dest = destination
        current_location = minimizer_dest
        sum_costs += minimizer_cost

    print 'node99'
    print
    print 'Cost: ', sum_costs


## Main loop

graph = read_graph()
```

```
M = 1e10
J = {}
for node in graph:
    J[node] = M
J['node99'] = 0

while 1:
    next_J = update_J(J, graph)
    if next_J == J:
        break
    else:
        J = next_J
print_best_path(J, graph)
```

A First Look at the Kalman Filter

Overview

This lecture provides a simple and intuitive introduction to the Kalman filter, for those who

- have heard of the Kalman filter but don't know how it works, or
- know the Kalman filter equations, but don't know where they come from.

For additional (more advanced) reading on the Kalman filter, see RMT3, section 2.7.

Required knowledge: Familiarity with matrix manipulations, multivariate normal distributions, covariance matrices, etc.

The Basic Idea

The Kalman filter has many applications in economics, but for now let's pretend that we are rocket scientists.

A missile has been launched from country Y and our mission is to track it.

Let $x \in \mathbb{R}^2$ denote the current location of the missile—a pair indicating latitude-longitute coordinates on a map

At the present moment in time, the precise location x is unknown, but we do have some beliefs about x

One way to summarize our knowledge is a point prediction $\hat{x}$

- But what if the president wants to know the probability that the missile is currently over the Sea of Japan?
- Better to summarize our beliefs with a bivariate density p
  - $\int_E p(x)dx$ indicates likelihood we attach to the missile being in region E

The density p is called our *prior* for the random variable x

To keep things tractable, we will always assume that our prior is Gaussian. In particular, we take

$$p = N(\hat{x}, \Sigma) \tag{1.13}$$

where $\hat{x}$ is the mean of the distribution and Σ is a 2×2 covariance matrix. In our simulations, we will suppose that

$$\hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \qquad \Sigma = \begin{pmatrix} 0.4 \\ 0.3 \\ 0.3 \\ 0.45 \end{pmatrix}$$

This density $p(x)$ is shown below as a contour map, with the center of the red ellipse being equal to $\hat{x}$

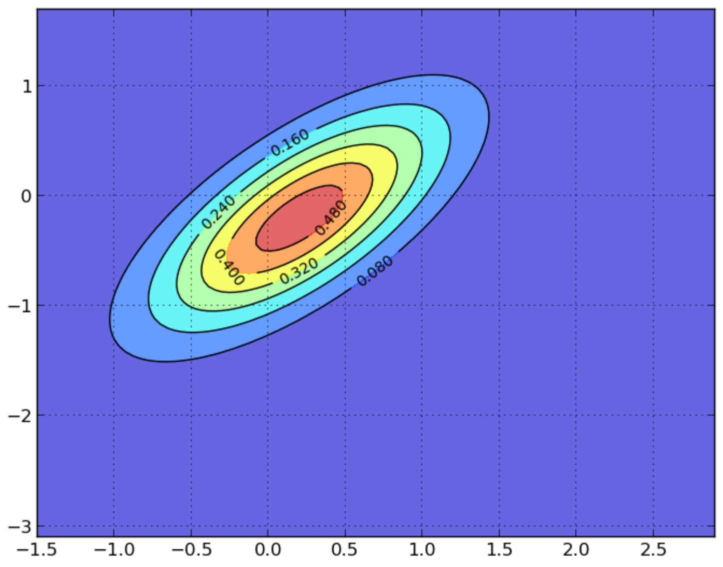


Figure 1.1: Prior density (Click this or any other figure to enlarge.)

The Filtering Step We are now presented with some good news and some bad news

The good news is that the missile has been located by our sensors, which report that the current location is $y = (2.3, -1.9)$

The next figure shows the original prior $p(x)$ and the new reported location y

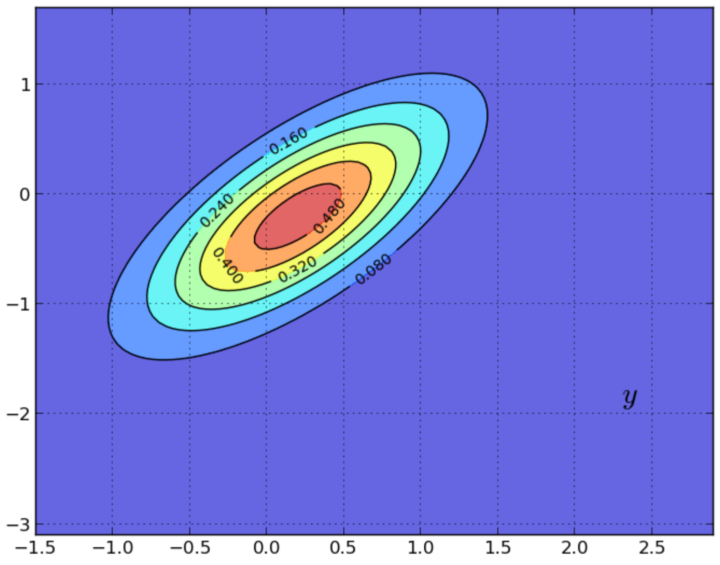
The bad news is that our sensors are imprecise.

In particular, we should interpret the output of our sensor not as $y = x$, but rather as

$$y = Gx + v, \quad \text{where} \quad v \sim N(0, R) \tag{1.14}$$

Here G and R are 2×2 matrices with R positive definite. Both are assumed known, and the shock v is assumed to be independent of x

How then should we combine our prior $p(x) = N(\hat{x}, \Sigma)$ and this new information y to improve our understanding of the location of the missile?



As you may have guessed, the answer is to use Bayes' theorem, which tells us we should update our prior $p(x)$ to $p(x \,|\, y)$ via

$$p(x \,|\, y) = \frac{p(y \,|\, x) \, p(x)}{p(y)}$$

In solving this for $p(x \,|\, y)$, we observe that

- $p(x) = N(\hat x, \Sigma)$
- In view of (1.14), the conditional density $p(y \,|\, x)$ is $N(Gx, R)$
- $p(y)$ does not depend on x, and enters into the calculations only as a normalizing constant

You can do the calculations yourself if you like, but they are quite fiddly, and since we are dealing with Gaussians and linear transformations, this is all well understood material

In particular, the solution is known [1] to be

$$p(x \,|\, y) = N(\hat x^F, \Sigma^F)$$

where

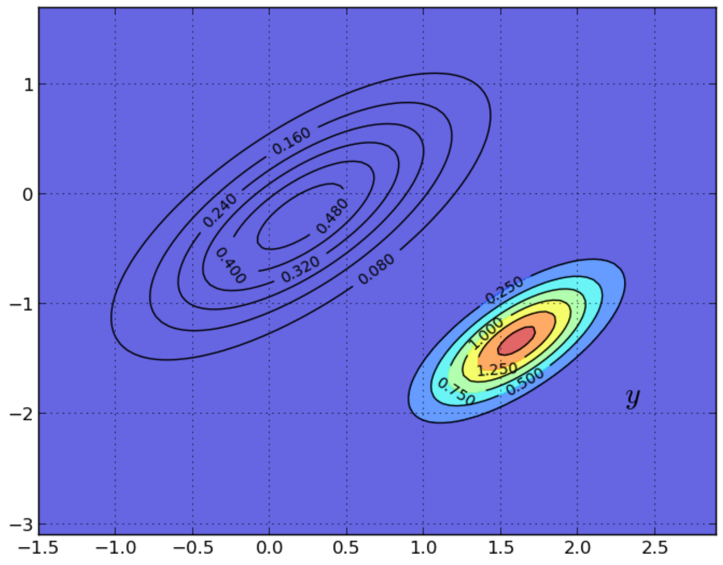
$$\hat x^F := \hat x + \Sigma G' (G \Sigma G' + R)^{-1}(y - G \hat x) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G' (G \Sigma G' + R)^{-1} G \Sigma \tag{1.15}$$

This new density $p(x \,|\, y) = N(\hat x^F, \Sigma^F)$ is shown in the next figure via contour lines and the color map

The original density is left in as contour lines for comparison

Our new density is a kind of "convex combination" of our prior $p(x)$ and our new information y

[1] See, for example, page 93 of C. M. Bishop (2006), *Pattern Recognition and Machine Learning*, Springer. To get from his expressions to the ones used above, you will also need to apply the Woodbury matrix identity.



The Forecast Step What have we achieved so far?

We have obtained probabilities for the current location of the state (missile) given prior and current information

This is called "filtering" rather than forecasting, because we are filtering out noise rather than looking into the future

- $p(x \,|\, y) = N(\hat{x}^F, \Sigma^F)$ is called the *filtering distribution*

But now let's suppose that we are given another task: To predict the location of the missile after one unit of time (whatever that may be) has elapsed

To do this we need a model of how the state evolves

Let's suppose that we have one, and that it's linear and Gaussian: In particular,

$$x_{t+1} = Ax_t + w_{t+1}, \quad \text{where} \quad w_t \sim N(0, Q) \tag{1.16}$$

Our aim is to combine this law of motion and our current distribution $p(x \,|\, y) = N(\hat{x}^F, \Sigma^F)$ to come up with a new *predictive* distribution for the location one unit of time hence

In view of (1.16), all we have to do is introduce a random vector $x^F \sim N(\hat{x}^F, \Sigma^F)$ and work out the distribution of $Ax^F + w$ where w is independent of x^F and has distribution $N(0, Q)$

Since linear combinations of Gaussians are Gaussian, $Ax^F + w$ is Gaussian

Elementary calculations and the expressions in (1.15) tell us that

$$\mathbb{E}[Ax^F + w] = A\mathbb{E}x^F + \mathbb{E}w = A\hat{x}^F = A\hat{x} + A\Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x})$$

and

$$\mathrm{Var}[Ax^F + w] = A\,\mathrm{Var}[x^F]A' + Q = A\Sigma^F A' + Q = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q$$

The matrix $A\Sigma G'(G\Sigma G' + R)^{-1}$ is often written as K_Σ and called the *Kalman gain*

- the subscript Σ has been added to remind us that it depends on Σ, but not y or $\hat{x}$

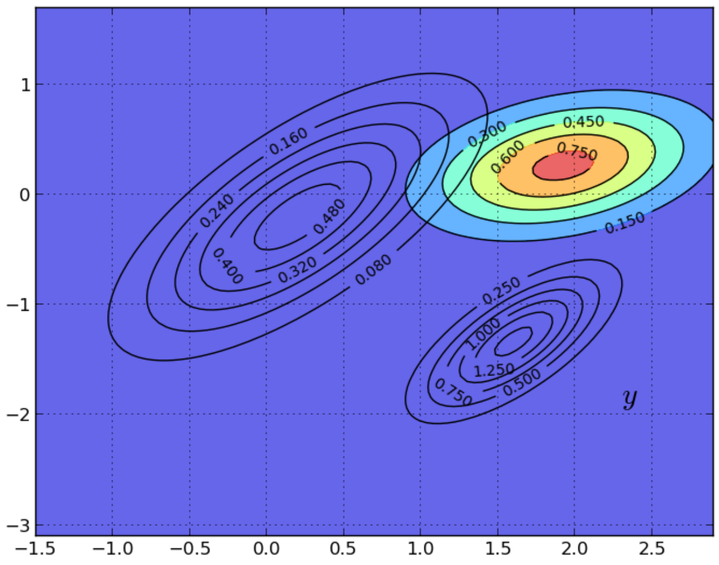
Using this notation, we can summarize our results as follows: Our updated prediction is the density $N(\hat{x}_{new}, \Sigma_{new})$ where

$$\begin{aligned} \hat{x}_{new} &:= A\hat{x} + K_\Sigma (y - G\hat{x}) \\ \Sigma_{new} &:= A\Sigma A' - K_\Sigma G \Sigma A' + Q \end{aligned} \tag{1.17}$$

- The density $p_{new}(x) = N(\hat{x}_{new}, \Sigma_{new})$ is called the *predictive distribution*

The predictive distribution is the new density shown in the following figure, where the update has used parameters

$$A = \begin{pmatrix} 1.2 \\ 0.0 \\ 0.0 \\ -0.2 \end{pmatrix}, \qquad Q = 0.3 * \Sigma$$



The code for producing this and all the preceding figures can be obtained `here`

The Recursive Procedure Let's look back at what we've done.

We started the current period with a prior $p(x)$ for the location of the missile

We then used the current measurement y to update to $p(x \mid y)$

Finally, we used the law of motion (1.16) for $\{x_t\}$ to update to $p_{new}(x)$

If we now step into the next period, we are ready to go round again, taking $p_{new}(x)$ as the current prior

Swapping notation $p_t(x)$ for $p(x)$ and $p_{t+1}(x)$ for $p_{new}(x)$, the full recursive procedure is:

1. Start the current period with prior $p_t(x) = N(\hat{x}_t, \Sigma_t)$

2. Observe current measurement y_t
3. Compute the filtering distribution $p_t(x \,|\, y) = N(\hat{x}_t^F, \Sigma_t^F)$ from $p_t(x)$ and y_t, applying Bayes rule and the conditional distribution (1.14)
4. Compute the predictive distribution $p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1})$ from the filtering distribution and (1.16)
5. Increment t by one and go to step 1

Repeating (1.17), the dynamics for $\hat{x}_t$ and Σ_t are as follows

$$\begin{aligned}\hat{x}_{t+1} &= A\hat{x}_t + K_{\Sigma_t}(y_t - G\hat{x}_t) \\ \Sigma_{t+1} &= A\Sigma_t A' - K_{\Sigma_t} G \Sigma_t A' + Q\end{aligned}$$

These are the standard dynamic equations for the Kalman filter. See, for example, RMT3, page 58.

Implementation: `kalman.py`

This section describes a module called `kalman` that implements the Kalman filter

- Code can be obtained from the main repository

In the module, the updating rules are wrapped up in a class, which bundles together

- all of the parameters (matrices, etc.) of a given model
- the moments $(\hat{x}_t, \Sigma_t)$ of the current prior
- a method `prior_to_filtered()` to update these moments to those of the filtering distribution $(\hat{x}_t^F, \Sigma_t^F)$
- a method `filtered_to_forecast()` to update the filtering distribution to the predictive distribution – which becomes the new prior $(\hat{x}_{t+1}, \Sigma_{t+1})$
- an `update()` method, which combines the last two methods

The program is simple and self-explanatory:

```python
import numpy as np
from numpy import dot
from numpy.linalg import inv

class Kalman:

    def __init__(self, A, G, Q, R):
        """
        Provide initial parameters describing the model.  All arguments should
        be Python scalars or NumPy ndarrays.

            * A is n x n
            * Q is n x n and positive definite
            * G is k x n
            * R is k x k and positive definite
        """
        self.A = np.array(A, dtype='float32')
        self.G = np.array(G, dtype='float32')
        self.Q = np.array(Q, dtype='float32')
        self.R = np.array(R, dtype='float32')

    def set_state(self, x_hat, Sigma):
        """
        Set the state, which is the mean x_hat and covariance matrix Sigma of
        the prior/predictive density.

            * x_hat is n x 1
            * Sigma is n x n and positive definite

        Must be Python scalars or NumPy arrays.
        """
        self.current_Sigma = np.array(Sigma, dtype='float32')
        self.current_x_hat = np.array(x_hat, dtype='float32')

    def prior_to_filtered(self, y):
        """
        Updates the moments (x_hat, Sigma) of the time t prior to the time t
        filtering distribution, using current measurement y_t.  The parameter
        y should be a Python scalar or NumPy array.  The updates are according
        to

            x_hat^F = x_hat + Sigma G' (G Sigma G' + R)^{-1}(y - G x_hat)
            Sigma^F = Sigma - Sigma G' (G Sigma G' + R)^{-1} G Sigma

        """
        # Simplify notation
        G, R = self.G, self.R
        x_hat, Sigma = self.current_x_hat, self.current_Sigma
        # And then update
        A = dot(Sigma, G.T)
        B = dot(dot(G, Sigma), G.T) + R
        if B.shape:  # If B has a shape, then it is multidimensional
            M = dot(A, inv(B))
        else:  # Otherwise it's just scalar
            M = A / B
        self.current_x_hat = x_hat + dot(M, (y - dot(G, x_hat)))
        self.current_Sigma = Sigma  - dot(M, dot(G,  Sigma))

    def filtered_to_forecast(self):
        """
        Updates the moments of the time t filtering distribution to the
        moments of the predictive distribution -- which becomes the time t+1
        prior
        """
        # Make local copies of names to simplify notation
        A, Q = self.A, self.Q
        x_hat, Sigma = self.current_x_hat, self.current_Sigma
        # And then update
        self.current_x_hat = dot(A, x_hat)
        self.current_Sigma = dot(A, dot(Sigma, A.T)) + Q

    def update(self, y):
        """
        Updates x_hat and Sigma given k x 1 ndarray y.  The full update, from
        one period to the next
        """
        self.prior_to_filtered(y)
        self.filtered_to_forecast()
```

Exercises

Exercise 1 Consider the following simple application of the Kalman filter, loosely based on RMT3, section 2.9.2

Suppose that

- all variables are scalars
- the hidden state $\{x_t\}$ is in fact constant, equal to some $\theta \in \mathbb{R}$ unknown to the modeler

State dynamics are therefore given by (1.16) with $A = 1$, $Q = 0$ and $x_0 = \theta$

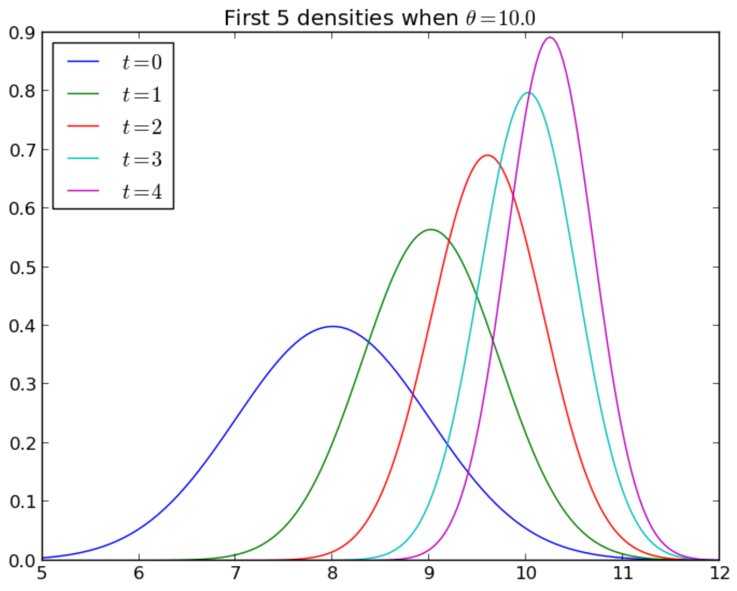
The measurement equation is $y_t = \theta + v_t$ where v_t is $N(0, 1)$ and iid

The task of this exercise to simulate the model and, using the module `kalman`, plot the first five predictive densities $p_t(x) = N(\hat{x}_t, \Sigma_t)$

As shown in RMT3, sections 2.9.1–2.9.2, these distributions asymptotically put all mass on the unknown value θ

In the simulation, take $\theta = 10$, $\hat{x}_0 = 8$ and $\Sigma_0 = 1$

Your figure should – modulo randomness – look something like this



Exercise 2 The preceding figure gives some support to the idea that probability mass converges to θ

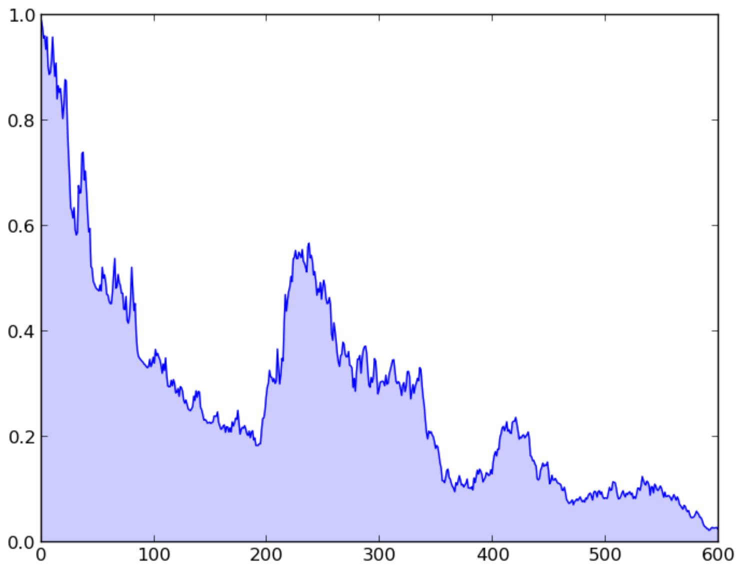
To get a better idea, choose a small $\epsilon > 0$ and calculate

$$z_t := 1 - \int_{\theta - \epsilon}^{\theta + \epsilon} p_t(x) dx$$

for $t = 0, 1, 2, \ldots, T$

Plot z_t against T, setting $\epsilon = 0.1$ and $T = 600$

Your figure should show error declining something like this, although the output is quite variable



Solutions

Solution to Exercise 1 To use `kalman` for this application, we need to set

```
theta = 10
A, G, Q, R = 1, 1, 0, 1
x_hat_0, Sigma_0 = 8, 1
```

We can then generate our draws of y_t via $\theta + N(0,1)$ and update the densities using these values

The code for doing this is below.

In the code, note the use of LaTeX expressions inside the figure labels and title. This requires proper integration of LaTeX and Matplotlib.

If you haven't set this up, then replace the LaTeX expressions with ordinary text

```
import numpy as np
import matplotlib.pyplot as plt
from kalman import Kalman
from scipy.stats import norm

## Parameters
theta = 10
A, G, Q, R = 1, 1, 0, 1
x_hat_0, Sigma_0 = 8, 1
## Initialize Kalman filter
kalman = Kalman(A, G, Q, R)
kalman.set_state(x_hat_0, Sigma_0)

N = 5
fig, ax = plt.subplots()
xgrid = np.linspace(theta - 5, theta + 2, 200)
for i in range(N):
    # Record the current predicted mean and variance, and plot their densities
    m, v = kalman.current_x_hat, kalman.current_Sigma
    m, v = float(m), float(v)
    ax.plot(xgrid, norm.pdf(xgrid, loc=m, scale=np.sqrt(v)), label=r'$t=%d$' % i)
    # Generate the noisy signal
    y = theta + norm.rvs(size=1)
    # Update the Kalman filter
    kalman.update(y)

ax.set_title(r'First %d densities when $\theta = %.1f$' % (N, theta))
ax.legend(loc='upper left')
fig.show()
```

Solution to Exercise 2 Borrowing code from the solution to exercise 1, we can solve exercise 2 as follows

```
import numpy as np
import matplotlib.pyplot as plt
from kalman import Kalman
from scipy.stats import norm
from scipy.integrate import quad

## Parameters
theta = 10
A, G, Q, R = 1, 1, 0, 1
x_hat_0, Sigma_0 = 8, 1
epsilon = 0.1
## Initialize Kalman filter
kalman = Kalman(A, G, Q, R)
kalman.set_state(x_hat_0, Sigma_0)

T = 600
z = np.empty(T)
for t in range(T):
    # Record the current predicted mean and variance, and plot their densities
    m, v = kalman.current_x_hat, kalman.current_Sigma
    m, v = float(m), float(v)
    f = lambda x: norm.pdf(x, loc=m, scale=np.sqrt(v))
    integral, error = quad(f, theta - epsilon, theta + epsilon)
    z[t] = 1 - integral
    # Generate the noisy signal and update the Kalman filter
    kalman.update(theta + norm.rvs(size=1))

fig, ax = plt.subplots()
ax.set_ylim(0, 1)
ax.set_xlim(0, T)
ax.plot(range(T), z)
ax.fill_between(range(T), np.zeros(T), z, color="blue", alpha=0.2)
fig.show()
```

Schelling's Segregation Model

Schelling's model studies the (in)stability of mixed neighborhoods

Schelling won the Nobel Prize for this (and other) work

Model shows how local interactions can lead to very interesting macroeconomic structure

In particular: Mild preferences can lead to strong segregation

What follows is a modified version that captures main idea

Outline of the Model

Suppose we have two types of people

- Orange people and green people

They live mixed together on the unit square

- Locations of the form (x, y), where $0 < x, y < 1$

Initially they are integrated

- Starting locations IID bivariate uniform on unit square

Each agent is now given the chance to stay or move

- Stay if they are "happy," move if they are "unhappy"
- Happy if half or more of 10 nearest neighbors are of the same type
- Nearest is in terms of Euclidean distance

If agent is unhappy

1. Draw random location
2. If happy at new location, move there
3. Else, go to step 1

We cycle through the agents, giving offers to move

Continuing until no-one wishes to move

Modeling

Agents are modeled as objects

- **Data:**
  - type and location
- **Methods:**
  - Determine whether happy or not given locations of other agents
  - **If not happy, move**
    * find a new location where happy

Pseudocode for the main loop

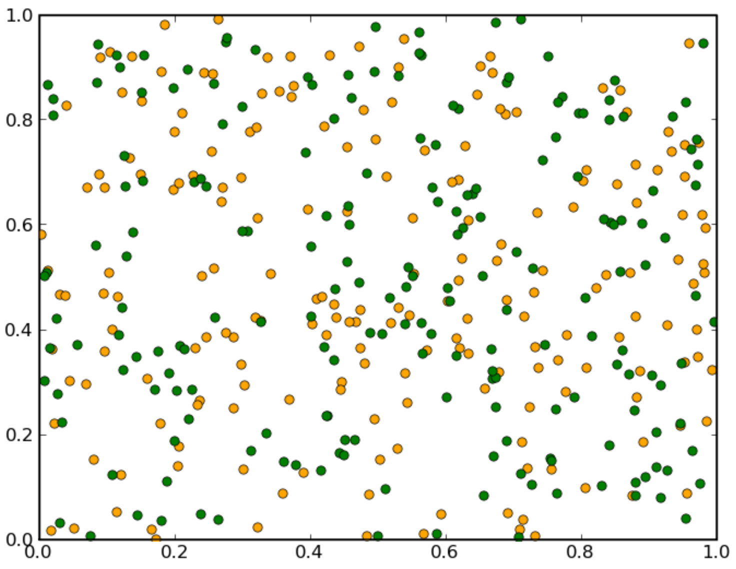
```
while agents are still moving:
    for agent in agents:
        give agent the opportunity to move
```

Use 200 agents of each type

Plot the before and after locations

My Results

Initially agents are randomly mixed



But after four rounds of the while loop they have become segregated

Conclusion

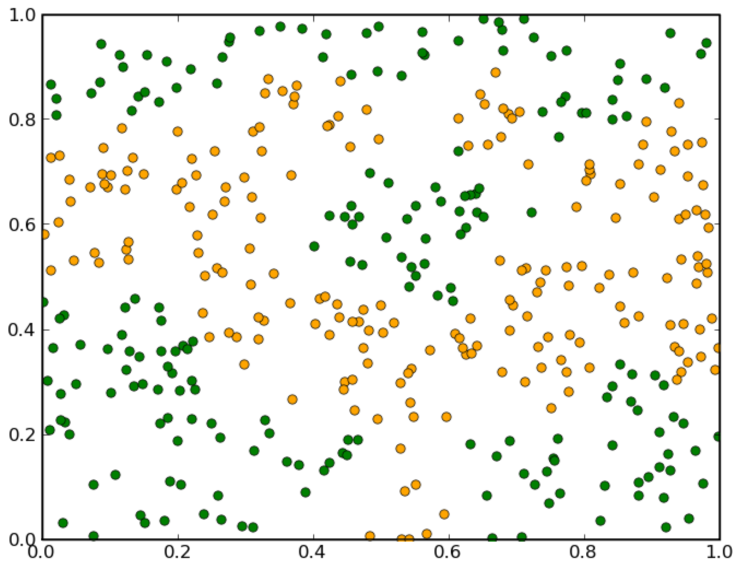
- **People in the model don't mind to live mixed with other type**
    - 50% of neighbors can be other color
- But even with these preferences, segregation still occurs

Exercise

Run your own simulation

Solution

```
## Filename: seg.py
## Author: John Stachurski

from random import uniform
from math import sqrt
import matplotlib.pyplot as plt

num_of_type_0 = 200
num_of_type_1 = 200
num_neighbors = 10      # Number of agents regarded as neighbors
require_same_type = 5   # Want at least this many neighbors to be same type
```



```python
class Agent:

    def __init__(self, type):
        self.type = type
        self.draw_location()

    def draw_location(self):
        self.location = uniform(0, 1), uniform(0, 1)

    def get_distance(self, other):
        "Computes euclidean distance between self and other agent."
        a = (self.location[0] - other.location[0])**2
        b = (self.location[1] - other.location[1])**2
        return sqrt(a + b)

    def happy(self, agents):
        "True if sufficient number of nearest neighbors are of the same type."
        distances = []
        # distances is a list of pairs (d, agent), where d is distance from
        # agent to self
        for agent in agents:
            if self != agent:
                distance = self.get_distance(agent)
                distances.append((distance, agent))
        # Sort from smallest to largest, according to distance
        distances.sort()
        # And extract the neighboring agents
        neighbors = [agent for d, agent in distances[:num_neighbors]]
        # Count how many neighbors have the same type as self
        num_same_type = sum(self.type == agent.type for agent in neighbors)
        return num_same_type >= require_same_type
    def update(self, agents):
        "If not happy, then randomly choose new locations until happy."
        while not self.happy(agents):
            self.draw_location()

def plot_distribution(agents, figname):
    "Plot the distribution of agents in file figname.png."
    x_values_0, y_values_0 = [], []
    x_values_1, y_values_1 = [], []
    for agent in agents:
        x, y = agent.location
        if agent.type == 0:
            x_values_0.append(x)
            y_values_0.append(y)
        else:
            x_values_1.append(x)
            y_values_1.append(y)
    plt.plot(x_values_0, y_values_0, 'o', markerfacecolor='orange', markersize=6)
    plt.plot(x_values_1, y_values_1, 'o', markerfacecolor='green', markersize=6)
    plt.savefig(figname)
    plt.clf()


def main():
    """
    Create a list of agents.  Loop until none wishes to move given the
    current distribution of locations.
    """
    agents = [Agent(0) for i in range(num_of_type_0)]
    agents.extend(Agent(1) for i in range(num_of_type_1))

    count = 1
    while 1:
        print 'Entering loop ', count
        plot_distribution(agents, 'fig%s.png' % count)
        count += 1
        no_one_moved = True
        for agent in agents:
            old_location = agent.location
            agent.update(agents)
            if agent.location != old_location:
                no_one_moved = False
        if no_one_moved:
            break
```

1.5 Part 4: Main Applications

Some info

1.5.1 Lectures

Modeling Career Choice

Overview

Next we study a computational problem concerning career and job choices. The model is originally due to Derek Neal [Neal1999] and this exposition draws on the presentation in RMT3, section 6.5.

Model features

- career and job within career both chosen to maximize expected discounted wage flow
- infinite horizon dynamic programming with two states variables

Model

In what follows we distinguish between a career and a job, where

- a *career* is understood to be a general field encompassing many possible jobs, and
- a *job* is understood to be a position with a particular firm

For workers, wages can be decomposed into the contribution of job and career

- $w_t = \theta_t + \epsilon_t$, where
  - θ_t is contribution of career at time t
  - ϵ_t is contribution of job at time t

At the start of time t, the worker has the following options

- retain their current (career, job) pair (θ_t, ϵ_t) — referred to hereafter as “stay put”
- retain their current career θ_t but redraw their job ϵ_t — referred to hereafter as “new job”
- redraw both their career θ_t and their job ϵ_t — referred to hereafter as “new life”

Draws of θ and ϵ are independent of each other and past values, with

- $\theta_t \sim F$
- $\epsilon_t \sim G$

Notice that the worker does not have the option to retain their job but redraw their career — starting a new career always requires starting a new job

A young worker aims to maximize the expected sum of discounted wages

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t \tag{1.18}$$

subject to the choice restrictions specified above

Let $V(\theta, \epsilon)$ denote the value function, which is the maximum of (1.18) over all feasible (career, job) policies, given the initial state (θ, ϵ)

The value function obeys

$$V(\theta, \epsilon) = \max\{I, II, III\},$$

where

$$I = \theta + \epsilon + \beta V(\theta, \epsilon) \tag{1.19}$$

$$II = \theta + \int \epsilon' G(d\epsilon') + \beta \int V(\theta, \epsilon') G(d\epsilon') \tag{1.20}$$

$$III = \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int V(\theta', \epsilon') G(d\epsilon') F(d\theta') \tag{1.21}$$

Evidently I, II and III correspond to "stay put", "new job" and "new life" respectively

Parameterization As in RMT3, section 6.5, we will focus on a discrete version of the model, parameterized as follows:

- both θ and ϵ take values in the set `np.linspace(0, B, N)` — an even grid of N points between 0 and B inclusive
- $N = 50$
- $B = 5$
- $\beta = 0.95$

The distributions F and G are discrete distributions generating draws from the grid points `np.linspace(0, B, N)`

A very useful family of discrete distributions is the Beta-binomial family, with probability mass function

$$p(k \,|\, n, a, b) = \binom{n}{k} \frac{B(k + a, n - k + b)}{B(a, b)}, \qquad k = 0, \ldots, n$$

Interpretation:

- draw q from a Beta distribution with shape parameters (a, b)
- run n independent binary trials, each with success probability q
- $p(k \,|\, n, a, b)$ is the probability of k successes in these n trials

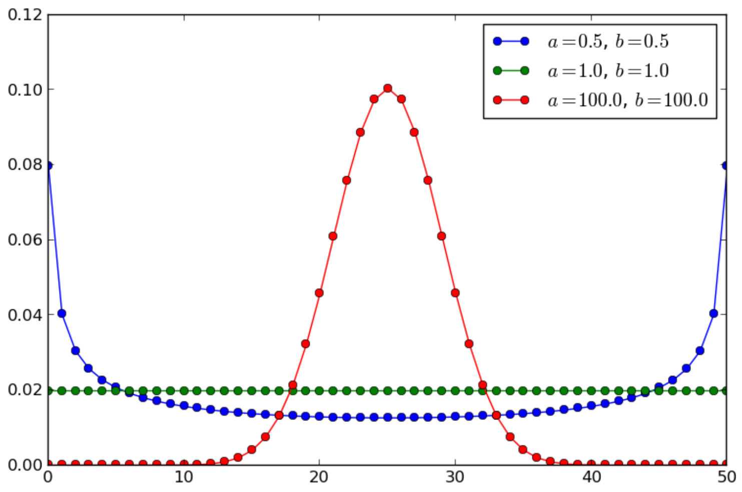
Nice properties:

- very flexible class of distributions, including uniform, symmetric unimodal, etc.
- only three parameters

Here's a figure showing the effect of different shape parameters when $n = 50$

The code that generated this figure is as follows

```python
from scipy.special import binom, beta
import matplotlib.pyplot as plt
import numpy as np

def gen_probs(n, a, b):
    probs = np.zeros(n+1)
```



```
    for k in range(n+1):
        probs[k] = binom(n, k) * beta(k + a, n - k + b) / beta(a, b)
    return probs

n = 50
a_vals = [0.5, 1, 100]
b_vals = [0.5, 1, 100]
fig, ax = plt.subplots()
for a, b in zip(a_vals, b_vals):
    ab_label = r'$a = %.1f$, $b = %.1f$' % (a, b)
    ax.plot(range(0, n+1), gen_probs(n, a, b), '-o', label=ab_label)
ax.legend()
fig.show()
```

Implementation: `career.py`

This section describes the module `career`, which solves the DP problem described above, and is provided in the main repository

The main aim of the module is to implement iteration with the Bellman operator T

In this model, T is defined by $Tv(\theta, \epsilon) = \max\{I, II, III\}$, where I, II and III are as given in (1.5.1), replacing V with v

The module `career` defines

- a function `gen_probs()` that computes Beta-binomial probabilities, as above
- a class `workerProblem` that encapsulates all the details of a particular parameterization
- a function `bellman()` that corresponds to the Bellman operator
- a function `get_greedy()` that computes policies from value functions

The code is as follows

```
import numpy as np
from scipy.special import binom, beta
```

```
def gen_probs(n, a, b):
    """
    Generate and return the vector of probabilities for the Beta-binomial
    (n, a, b) distribution.
    """
    probs = np.zeros(n+1)
    for k in range(n+1):
        probs[k] = binom(n, k) * beta(k + a, n - k + b) / beta(a, b)
    return probs


class workerProblem:

    def __init__(self, B=5.0, beta=0.95, N=50, F_a=1, F_b=1, G_a=1, G_b=1):
        self.beta, self.N, self.B = beta, N, B
        self.theta = np.linspace(0, B, N)      # set of theta values
        self.epsilon = np.linspace(0, B, N)    # set of epsilon values
        self.F_probs = gen_probs(N-1, F_a, F_b)
        self.G_probs = gen_probs(N-1, G_a, G_b)
        self.F_mean = np.sum(self.theta * self.F_probs)
        self.G_mean = np.sum(self.epsilon * self.G_probs)

def bellman(w, v):
    """
    The Bellman operator.

        * w is an instance of workerProblem
        * v is a 2D NumPy array representing the value function

    The array v should be interpreted as v[i, j] = v(theta_i, epsilon_j).
    Returns the updated value function Tv as an array of shape v.shape
    """
    new_v = np.empty(v.shape)
    for i in range(w.N):
        for j in range(w.N):
            v1 = w.theta[i] + w.epsilon[j] + w.beta * v[i, j]
            v2 = w.theta[i] + w.G_mean + w.beta * np.dot(v[i, :], w.G_probs)
            v3 = w.G_mean + w.F_mean + w.beta * \
                    np.dot(w.F_probs, np.dot(v, w.G_probs))
            new_v[i, j] = max(v1, v2, v3)
    return new_v

def get_greedy(w, v):
    """
    Compute optimal actions taking v as the value function.  Parameters are
    the same as for bellman().  Returns a 2D NumPy array "policy", where
    policy[i, j] is the optimal action at state (theta_i, epsilon_j).  The
    optimal action is represented as an integer in the set 1, 2, 3, where 1 =
    'stay put', 2 = 'new job' and 3 = 'new life'
    """
    policy = np.empty(v.shape, dtype=int)
    for i in range(w.N):
        for j in range(w.N):
            v1 = w.theta[i] + w.epsilon[j] + w.beta * v[i, j]
            v2 = w.theta[i] + w.G_mean + w.beta * np.dot(v[i, :], w.G_probs)
            v3 = w.G_mean + w.F_mean + w.beta * \
                    np.dot(w.F_probs, np.dot(v, w.G_probs))
            if v1 > max(v2, v3):
                action = 1
            elif v2 > max(v1, v3):
                action = 2
            else:
                action = 3
            policy[i, j] = action
    return policy
```

The default probability distributions in `workerProblem` correspond to discrete uniform distributions (see *the Beta-binomial figure*)

In fact all our default settings correspond to the version studied in RMT3, section 6.5.

Hence we can reproduce figures 6.5.1 and 6.5.2 shown there, which exhibit the value function and optimal policy respectively

Here's the value function

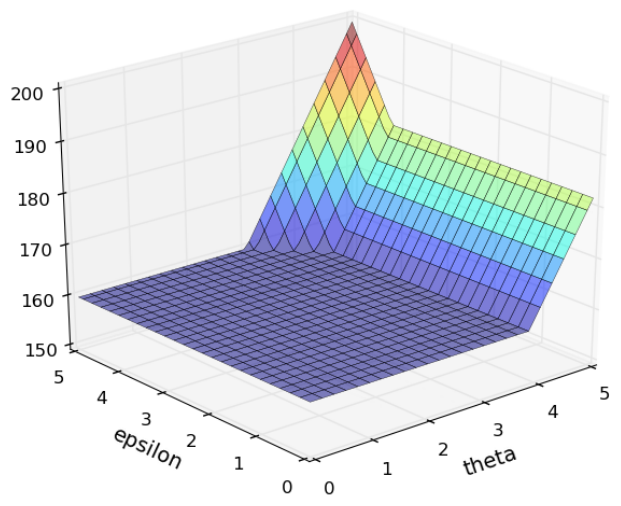


Figure 1.2: Value function with uniform probabilities

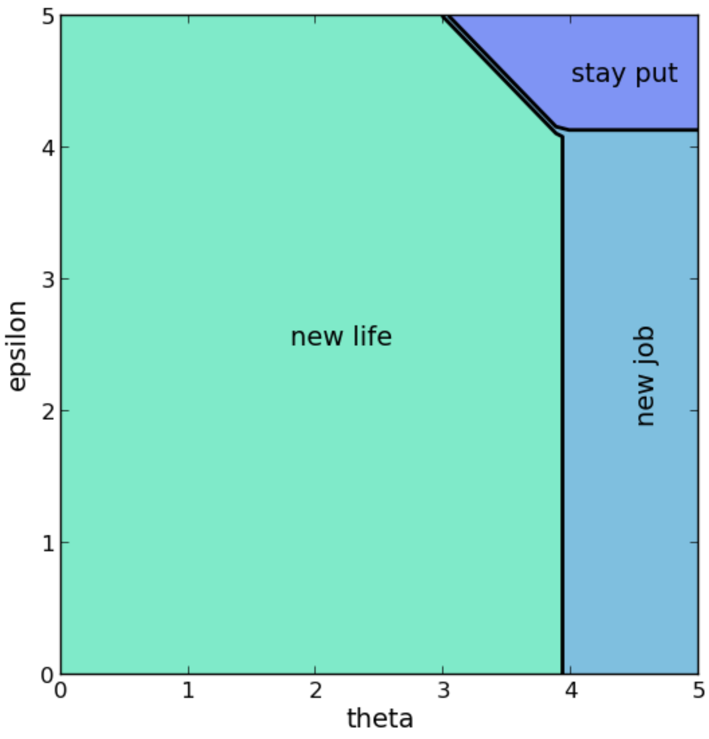
The code used to produce this plot was

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm
from career import *
from compute_fp import compute_fixed_point

wp = workerProblem()
v_init = np.ones((wp.N, wp.N))*100
v = compute_fixed_point(bellman, wp, v_init)

fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111, projection='3d')
```

```
tg, eg = np.meshgrid(wp.theta, wp.epsilon)
ax.plot_surface(tg, eg, v.T, rstride=2, cstride=2, cmap=cm.jet, alpha=0.5,
        linewidth=0.25)
ax.set_zlim(150, 200)
ax.set_xlabel('theta', fontsize=14)
ax.set_ylabel('epsilon', fontsize=14)
fig.show()
```

The code pulls in the convenience function `compute_fixed_point()` from the module `compute_fp`, which can be found in the main repository

The optimal policy can be represented as follows (see *Exercise 3* for code)



Interpretation:

- If both job and career are poor or mediocre, the worker will experiment with new job and new career
- If career is sufficiently good, the worker will hold it and experiment with new jobs until a sufficiently good one is found
- If both job and career are good, the worker will stay put

Notice that the worker will always holds on to a sufficiently good career, but not necessarily hold on to even the best paying job

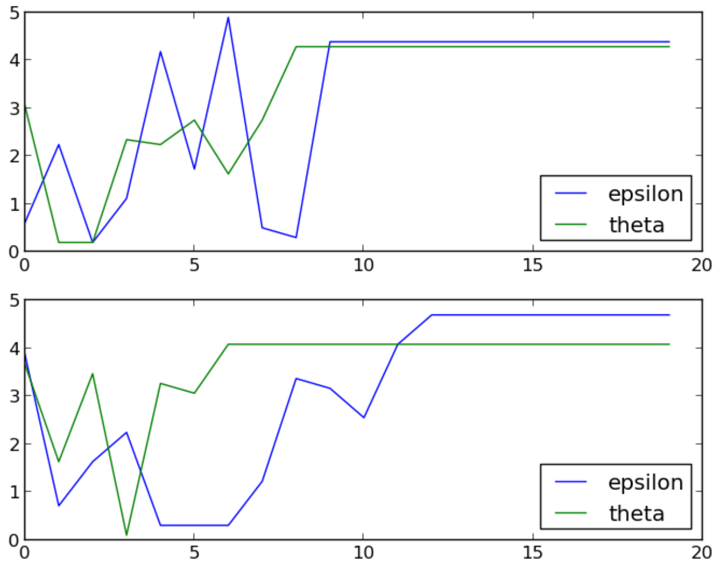
The reason is that high lifetime wages require both variables to be large, and the worker cannot change careers without changing jobs

- Sometimes a good job must be sacrificed in order to change to a better career

Exercises

Exercise 1 Using the default parameterization in the class `workerProblem`, generate and plot typical sample paths for θ and ϵ when the worker follows the optimal policy

In particular, modulo randomness, reproduce the following figure (where the horizontal axis represents time)



Hint: To generate the draws from the distributions F and G, use the module `discreterv`, which can be found in the main repository

Solution: *View solution*

Exercise 2 Let's now consider how long it takes for the worker to settle down to a permanent job, given a starting point of $(\theta, \epsilon) = (0, 0)$

In other words, we want to study the distribution of the random variable

$$T^* := \text{the first point in time from which the worker's job no longer changes}$$

Evidently, the worker's job becomes permanent if and only if (θ_t, ϵ_t) enters the "stay put" region of (θ, ϵ) space

Letting S denote this region, T^* can be expressed as the first passage time to S under the optimal policy:

$$T^* := \inf\{t \geq 0 \,|\, (\theta_t, \epsilon_t) \in S\}$$

Collect 25,000 draws of this random variable and compute the median (which should be about 7)

Repeat the exercise with $\beta = 0.99$ and interpret the change

Solution: *View solution*

Exercise 3 As best you can, reproduce *the figure showing the optimal policy*

Hint: The `get_greedy()` function returns a representation of the optimal policy where values 1, 2 and 3 correspond to "stay put", "new job" and "new life" respectively. Use this and `contourf` from `matplotlib.pyplot` to produce the different shadings.

Now set `G_a = G_b = 100` and generate a new figure with these parameters. Interpret.

Solution: *View solution*

References

Optimal Savings

Overview

Next we study the standard optimal savings problem for an infinitely lived consumer—the "common ancestor" described in RMT3, section 1.3

- Also known as the income fluctuation problem
- An important sub-problem for many representative macroeconomic models
  - [Aiyagari1994]
  - [Huggett1993]
  - etc.
- Useful references include [Deaton1991], [DenHaan2010], [Kuhn2013], [Rabault2002], [Reiter2008] and [SchechtmanEscudero1977]

Our presentation of the model will be relatively brief

- For further details on economic intuition, implication and models, see RMT3
- Proofs of all mathematical results stated below can be found in `this paper`

In this lecture we will explore an alternative to value function iteration (VFI) called *policy function iteration* (PFI)

- Based on the Euler equation, and not to be confused with Howard's policy iteration algorithm
- Globally convergent under mild assumptions, even when utility is unbounded (both above and below)
- Numerically, turns out to be faster and more efficient than VFI for this model

Model features

- Infinite horizon dynamic programming with two states and one control

The Optimal Savings Problem

Consider a household that chooses a state-contingent consumption plan $\{c_t\}_{t \geq 0}$ to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$c_t + a_{t+1} \leq Ra_t + z_t, \qquad c_t \geq 0, \qquad a_t \geq -b \qquad t = 0, 1, \ldots \tag{1.22}$$

Here

- $\beta \in (0, 1)$ is the discount factor
- a_t is asset holdings at time t, with ad-hoc borrowing constraint $a_t \geq -b$
- c_t is consumption
- z_t is non-capital income (wages, unemployment compensation, etc.)
- $R := 1 + r$, where $r > 0$ is the interest rate on savings

Assumptions

1. $\{z_t\}$ is a finite Markov process with Markov matrix Π taking values in Z
2. $|Z| < \infty$ and $Z \subset (0, \infty)$
3. $r > 0$ and $\beta R < 1$
4. u is smooth, strictly increasing and strictly concave with $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$

The asset space is $[-b, \infty)$ and the state is the pair $(a, z) \in S := [-b, \infty) \times Z$

A *feasible consumption path* from $(a, z) \in S$ is a consumption sequence $\{c_t\}$ such that $\{c_t\}$ and its induced asset path $\{a_t\}$ satisfy

1. $(a_0, z_0) = (a, z)$
2. the feasibility constraints in (1.22), and
3. measurability of c_t w.r.t. the filtration generated by $\{z_1, \ldots, z_t\}$

The meaning of the third point is just that consumption at time t can only be a function of outcomes that have already been observed

The *value function* $V \colon S \to \mathbb{R}$ is defined by

$$V(a, z) := \sup \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \tag{1.23}$$

where the supremum is over all feasible consumption paths from (a, z).

An *optimal consumption path* from (a, z) is a feasible consumption path from (a, z) that attains the supremum in (1.23)

Given our assumptions, it is `known` that

1. For each $(a, z) \in S$, a unique optimal consumption path from (a, z) exists
2. This path is the unique feasible path from (a, z) satisfying the Euler equality

$$u'(c_t) = \max \left\{ \beta R \, \mathbb{E}_t [u'(c_{t+1})] \, , \; u'(Ra_t + z_t + b) \right\} \tag{1.24}$$

and the transversality condition

$$\lim_{t \to \infty} \beta^t \, \mathbb{E} \left[u'(c_t) a_{t+1} \right] = 0. \tag{1.25}$$

Moreover, there exists an *optimal consumption function* $c^* : S \to [0, \infty)$ such that the path from (a, z) generated by

$$(a_0, z_0) = (a, z), \quad z_{t+1} \sim \Pi(z_t, dy), \quad c_t = c^*(a_t, z_t) \quad \text{and} \quad a_{t+1} = R a_t + z_t - c_t$$

satisfied both (1.24) and (1.25), and hence is the unique optimal path from (a, z)

In summary, to solve the optimization problem, we need to compute c^*

Computation

There are two standard ways to solve for c^*

1. Value function iteration (VFI)
2. Policy function iteration (PFI) using the Euler inequality

Policy function iteration

We can rewrite (1.24) to make it a statement about functions rather than random variables

In particular, consider the functional equation

$$u' \circ c\,(a, z) = \max \left\{ \gamma \int u' \circ c \, \{Ra + z - c(a, z), \, \acute{z}\} \, \Pi(z, d\acute{z}) \, , \; u'(Ra + z + b) \right\} \tag{1.26}$$

where $\gamma := \beta R$ and $u' \circ c(s) := u'(c(s))$

Equation (1.26) is a functional equation in c

In order to identify a solution, let $\mathscr{C}$ be the set of candidate consumption functions $c \colon S \to \mathbb{R}$ such that

- each $c \in \mathscr{C}$ is continuous and (weakly) increasing
- $\min Z \leq c(a, z) \leq Ra + z + b$ for all $(a, z) \in S$

In addition, let $K : \mathscr{C} \to \mathscr{C}$ be defined as follows:

For given $c \in \mathscr{C}$, the value $Kc(a, z)$ is the unique $t \in J(a, z)$ that solves

$$u'(t) = \max \left\{ \gamma \int u' \circ c \, \{Ra + z - t, \, \acute{z}\} \, \Pi(z, d\acute{z}) \, , \; u'(Ra + z + b) \right\} \tag{1.27}$$

where

$$J(a, z) := \{ t \in \mathbb{R} \, : \, \min Z \leq t \leq Ra + z + b \} \tag{1.28}$$

We refer to K as Coleman's policy function operator [Coleman1990]

It is `known` that

- K is a contraction mapping on $\mathscr{C}$ under the metric

$$\rho(c, d) := \| \, u' \circ c - u' \circ d \, \| := \sup_{s \in S} | \, u'(c(s)) - u'(d(s)) \, | \qquad (c, d \in \mathscr{C})$$

- The metric ρ is complete on $\mathscr{C}$

- Convergence in ρ implies uniform convergence on compacts

In consequence, K has a unique fixed point $c^* \in \mathscr{C}$ and $K^n c \to c^*$ as $n \to \infty$ for any $c \in \mathscr{C}$

By the definition of K, the fixed points of K in $\mathscr{C}$ coincide with the solutions to (1.26) in $\mathscr{C}$

In particular, it `can be shown` that the path $\{c_t\}$ generated from $(a_0, z_0) \in S$ using policy function c^* is the unique optimal path from $(a_0, z_0) \in S$

TL;DR The unique optimal policy can be computed by picking any $c \in \mathscr{C}$ and iterating with the operator K defined in (1.27)

Value function iteration

The Bellman operator for this problem is given by

$$Tv(a, z) = \max_{0 \leq c \leq Ra+z+b} \left\{ u(c) + \beta \int v(Ra + z - c, \acute{z}) \Pi(z, d\acute{z}) \right\} \tag{1.29}$$

We have to be careful with VFI (i.e., iterating with T) in this setting because u is not assumed to be bounded

- In fact typically unbounded both above and below — e.g. $u(c) = \log c$
- In which case, the standard DP theory does not apply
- $T^n v$ not guaranteed to converge to the value function for arbitrary continous bounded v

Nonetheless, we can always try the strategy "iterate and hope"

- In this case we can check the outcome by comparing with PFI
- The latter is known to converge, as described above

Implementation The following code provides implementations of both VFI and PFI

- file `ifp.py`, provided in the main repository

Description and clarifications are given below

```
"""
Origin: QEwP by John Stachurski and Thomas J. Sargent
Date: 3/2013
File: ifp.py

Functions for solving the income fluctuation problem. Iteration with either
the Coleman or Bellman operators from appropriate initial conditions leads to
convergence to the optimal consumption policy.  The income process is a finite
state Markov chain.  Note that the Coleman operator is the preferred method,
as it is almost always faster and more accurate.  The Bellman operator is only
provided for comparison.

"""

import numpy as np
from scipy.optimize import fminbound, brentq
from scipy import interp

class consumerProblem:
    """
    This class is just a "struct" to hold the collection of parameters
    defining the consumer problem.
    """
    def __init__(self,
            r=0.01,
            beta=0.96,
            Pi=((0.6, 0.4), (0.05, 0.95)),
            z_vals=(0.5, 1.0),
            b=0,
            grid_max=16,
            grid_size=50,
            u=np.log,
            du=lambda x: 1/x):
        """
        Parameters:

            * r and beta are scalars with r > 0 and (1 + r) * beta < 1
            * Pi is a 2D NumPy array --- the Markov matrix for {z_t}
            * z_vals is an array/list containing the state space of {z_t}
            * u is the utility function and du is the derivative
            * b is the borrowing constraint
            * grid_max and grid_size describe the grid used in the solution

        """
        self.u, self.du = u, du
        self.r, self.R = r, 1 + r
        self.beta, self.b = beta, b
        self.Pi, self.z_vals = np.array(Pi), tuple(z_vals)
        self.asset_grid = np.linspace(-b, grid_max, grid_size)


def bellman_operator(cp, V, return_policy=False):
    """
    The approximate Bellman operator, which computes and returns the updated
    value function TV (or the V-greedy policy c if return_policy == True).

    Parameters:

        * cp is an instance of class consumerProblem
        * V is a NumPy array of dimension len(cp.asset_grid) x len(cp.z_vals)

    """
    R, Pi, beta, u, b = cp.R, cp.Pi, cp.beta, cp.u, cp.b  # Simplify names
    asset_grid, z_vals = cp.asset_grid, cp.z_vals          # Simplify names
    new_V = np.empty(V.shape)
    new_c = np.empty(V.shape)
    # Turn V into a function based on linear interpolation along the asset grid
    vf = lambda a, i_z: interp(a, asset_grid, V[:, i_z])
    z_index = range(len(z_vals))

    for i_a, a in enumerate(asset_grid):
        for i_z, z in enumerate(z_vals):
            def obj(c):  # Define objective function to be *minimized*
                y = sum(vf(R * a + z - c, j) * Pi[i_z, j] for j in z_index)
                return - u(c) - beta * y
            c_star = fminbound(obj, np.min(z_vals), R * a + z + b)
            new_c[i_a, i_z], new_V[i_a, i_z] = c_star, -obj(c_star)

    if return_policy:
        return new_c
    else:
        return new_V


def coleman_operator(cp, c):
    """
    The approximate Coleman operator.  Iteration with this operator
    corresponds to policy function iteration.  Computes and returns the
    updated consumption policy c.

    Parameters:

        * cp is an instance of class consumerProblem
        * c is a NumPy array of dimension len(cp.asset_grid) x len(cp.z_vals)

    The array c is replaced with a function cf that implements univariate
    linear interpolation over the asset grid for each possible value of z.
    """
    R, Pi, beta, du, b = cp.R, cp.Pi, cp.beta, cp.du, cp.b  # Simplify names
    asset_grid, z_vals = cp.asset_grid, cp.z_vals           # Simplify names
    z_size = len(z_vals)
    gamma = R * beta
    vals = np.empty(z_size)  # Empty array used below
    def cf(a):
        """
        The call cf(a) returns an array containing the values c(a, z) for each
        z in z_vals.  For each such z, the value c(a, z) is constructed by
        univariate linear approximation over asset space, based on the values
        in the array c
        """
        for i in range(z_size):
            vals[i] = interp(a, cp.asset_grid, c[:, i])
        return vals

    Kc = np.empty(c.shape)
    for i_a, a in enumerate(asset_grid):
        for i_z, z in enumerate(z_vals):
            def h(t):
                expectation = np.dot(du(cf(R * a + z - t)), Pi[i_z, :])
                return du(t) - max(gamma * expectation, du(R * a + z + b))
            Kc[i_a, i_z] = brentq(h, np.min(z_vals), R * a + z + b)

    return Kc

def initialize(cp):
    """
    Creates a suitable initial conditions V and c for value function and
    policy function iteration respectively.

        * cp is an instance of class consumerProblem.

    """
    R, beta, u, b = cp.R, cp.beta, cp.u, cp.b             # Simplify names
    asset_grid, z_vals = cp.asset_grid, cp.z_vals         # Simplify names
    shape = len(asset_grid), len(z_vals)
    V, c = np.empty(shape), np.empty(shape)
    for i_a, a in enumerate(asset_grid):
        for i_z, z in enumerate(z_vals):
            c_max = R * a + z + b
            c[i_a, i_z] = c_max
            V[i_a, i_z] = u(c_max) / (1 - beta)
    return V, c
```

The code contains the following definitions

- class *consumerProblem*, which produces struct-type objects that contain all the relevant parameters of a given model
- function *bellman_operator()*, which implements the Bellman operator T specified above
- function *coleman_operator()*, which implements the Coleman operator K specified above
- function *initialize()*, which generates suitable initial conditions for iteration

The functions *bellman_operator()* and *coleman_operator()* both use linear interpolation along the asset grid to approximate the value and consumption functions

The following exercises walk you through several applications where policy functions are computed

In exercise 1 you will see that while VFI and PFI produce similar results, the latter is much faster

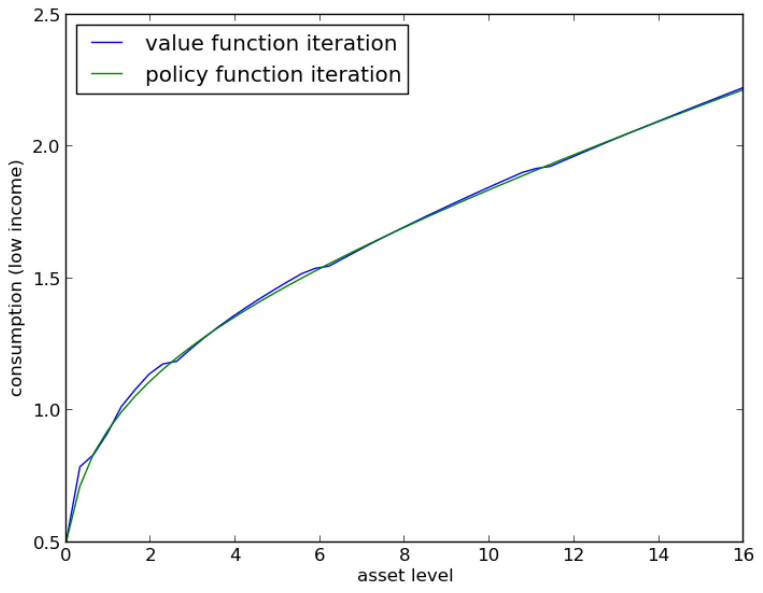
- Because we are exploiting analytically derived first order conditions

Another benefit of working in policy function space rather than value function space is that value functions typically have more curvature

- Makes them harder to approximate numerically

Exercises

Exercise 1 The first exercise is to replicate the following figure, which compares PFI and VFI as solution methods



The figure shows consumption policies computed by iteration of K and T respectively

- In the case of iteration with T, the final value function is used to compute the observed policy

Consumption is shown as a function of assets with income z held fixed at its smallest value

The following details are needed to replicate the figure

- The parameters are the default parameters in the definition of *consumerProblem*
- The initial conditions are the default ones from *initialize()*
- Both operators are iterated 80 times

When you run your code you will observe that iteration with K is faster than iteration with T

If you are using *IPython*, a comparison of the operators can be made as follows

```
In [12]: run ifp

In [13]: cp = consumerProblem()

In [14]: v, c = initialize(cp)

In [15]: timeit bellman_operator(cp, v)
10 loops, best of 3: 157 ms per loop

In [16]: timeit coleman_operator(cp, c)
10 loops, best of 3: 29.5 ms per loop
```

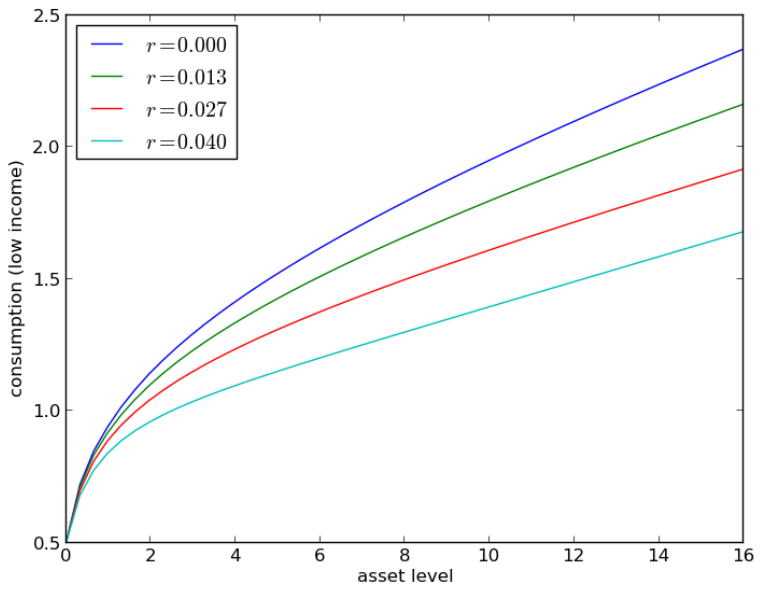
The output shows that Coleman operator is about 5 times faster

From now on we will only use the Coleman operator

Solution: *View solution*

Exercise 2 Next let's consider how the interest rate affects consumption

Reproduce the following figure, which shows (approximately) optimal consumption policies for different interest rates



- Other than r, all parameters are at their default values
- r steps through *np.linspace(0, 0.04, 4)*
- Consumption is plotted against assets for income shock fixed at the smallest value

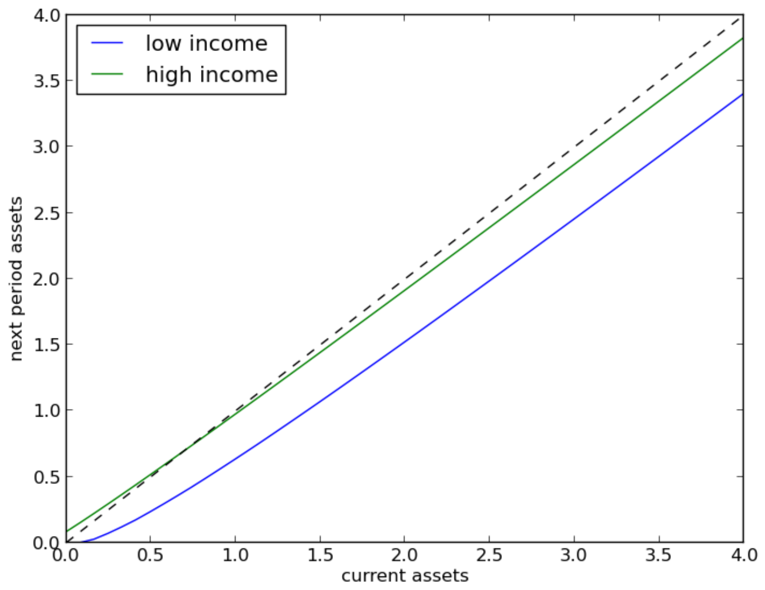
The figure shows that higher interest rates boost savings and hence suppress consumption

Solution: *View solution*

Exercise 3 Now let's consider the long run asset levels held by households

We'll take r = *0.03* and otherwise use default parameters

The following figure is a 45 degree diagram showing the law of motion for assets when consumption is optimal



The green line and blue line represent the function

$$a' = h(a, z) := Ra + z - c^*(a, z)$$

when income z takes its high and low values repectively

The dashed line is the 45 degree line

We can see from the figure that the dynamics will be stable — assets do not diverge

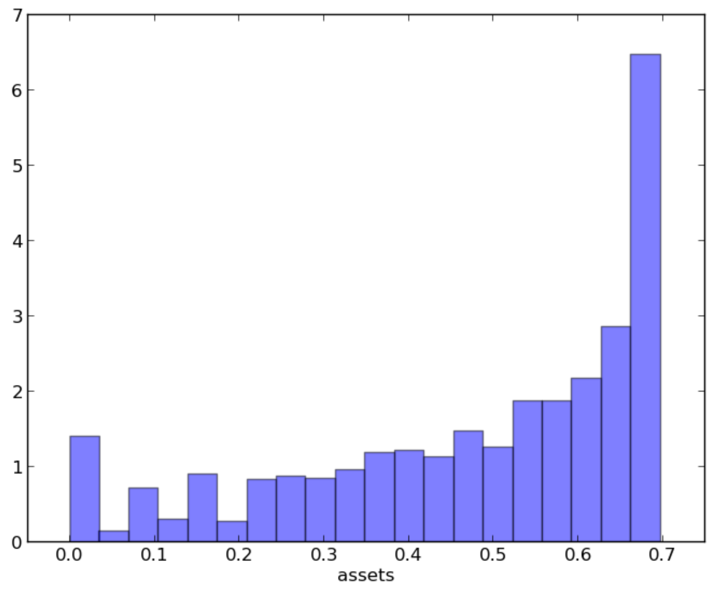
In fact there is a unique stationary distribution of assets that we can calculate by simulation

- Can be proved via theorem 2 of [HopenhaynPrescott1992]
- Represents the long run dispersion of assets across households when households have idiosyncratic shocks

Ergodicity is valid here, so stationary probabilities can be calculated by averaging over a single long time series

- Hence to approximate the stationary distribution we can simulate a long time series for assets and histogram, as in the following figure

Your task is to replicate the figure



- Parameters are as discussed above
- The histogram in the figure used a single time series $\{a_t\}$ of length 500,000
- Given the length of this time series, the initial condition (a_0, z_0) will not matter
- You might find it helpful to use the module *mc_sample* in the main repository
- Note that the simulations will be relatively slow due to the inherent need for loops — we'll talk about how to speed up this kind of code a bit later on

Solution: *View solution*

Exercise 4 Following on from exercises 2 and 3, let's look at how savings and aggregate asset holdings vary with the interest rate

- Note: RMT3 section 18.6 can be consulted for more background on the topic treated in this exercise

For a given parameterization of the model, the mean of the stationary distribution can be interpreted as aggregate capital in an economy with a unit mass of *ex-ante* identical households facing idiosyncratic shocks

Let's look at how this measure of aggregate capital varies with the interest rate and borrowing constraint

The next figure plots aggregate capital against the interest rate for *b in (1, 3)*

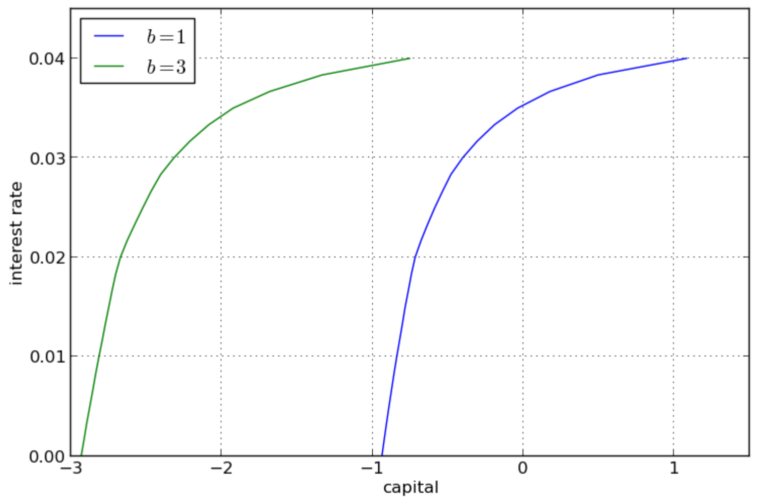
As is traditional, the price (interest rate) is on the vertical axis

The horizontal axis is aggregate capital computed as the mean of the stationary distribution

Exercise 4 is to replicate the figure, making use of code from previous exercises

Try to explain why the measure of aggregate capital is equal to $-b$ when $r = 0$ for both cases shown here

Solution: *View solution*



References

Linear Stochastic Models

Overview

In this lecture we study stochastic dynamic models that are linear in lagged values and shocks

We focus in particular on the class of processes that can be described by stationary ARMA(p, q) models

This class of models is

- extremely broad in terms of the kinds of dynamics it can represent, and yet
- simple enough to be described by an elegant and insightful theory

We consider the dynamics of these models in both the time and frequency domain

Prerequisite knowledge consists of basic probability and analysis. For supplementary reading, see

- RMT3, chapter 2
- [Sargent1987], chapter 11
- [Shiryaev1995], chapter 6
- [CryerChan2008], all

Covariance Stationary Processes

In this lecture we consider sequences of random variables $\{X_t\}$ taking values in $\mathbb{R}$

Following most of the literature in this field, we take t to be in $\mathbb{Z}$, the set of integers, so that

$$\{X_t\} = \dots, X_{-2}, X_{-1}, X_0, X_1, X_2, \dots$$

In other words, $\{X_t\}$ begins in the infinite past and extends to the infinite future

The kinds of processes we are most interested in are processes that are in some sense *stationary*, or become stationary after some transformation (differencing, cointegration, etc.)

Loosely speaking, stationarity means that the underlying probabilistic structure generating the process is invariant over time

It is natural to seek models that are stationary under some transformation, because successful modeling typically requires identifying deep structure that is relatively constant

If such structure can be found, then each new observation $X_t, X_{t+1}, \ldots$ provides additional information about it — which is how we learn from data

In this lecture we will focus on so-called "wide-sense" or "covariance" stationary processes

Definitions The process $\{X_t\}$ is called *covariance stationary* if

1. Its mean $\mu := \mathbb{E} X_t$ does not depend on t
2. For all k in $\mathbb{Z}$, the k-th autocovariance $\gamma(k) := \mathbb{E}(X_t - \mu)(X_{t+k} - \mu)$ is finite and depends only on k

The function $\gamma \colon \mathbb{Z} \to \mathbb{R}$ is called the *autocovariance function* of the process

In this lecture we will focus exclusively on covariance stationary processes

We will also assume that $\mu = 0$, since working with non-zero mean processes involves no more than adding a constant

Examples Perhaps the simplest class of covariance stationary processes is the white noise processes

A process $\{\epsilon_t\}$ is called a *white noise process* if

1. $\mathbb{E} X_t = 0$
2. $\gamma(k) = \sigma^2 1\{k = 0\}$ for some $\sigma > 0$

Here $1\{k = 0\}$ is defined to be 1 if $k = 0$ and zero otherwise

Evidently a white noise process is covariance stationary From this simple building block we can construct a very flexible family of covariance stationary processes called the *generalized linear processes*

This term refers to the set of processes $\{X_t\}$ that can be written as

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}, \qquad t \in \mathbb{Z} \tag{1.30}$$

where $\{\psi_t\}$ is a real sequence with $\sum_{t=0}^{\infty} \psi_t^2 < \infty$ and $\{\epsilon_t\}$ is white noise

With a few manipulations you will be able to confirm that the autocovariance function for this generalized linear process is

$$\gamma(k) = \sigma^2 \sum_{j=0}^{\infty} \psi_j \psi_{j+k}$$

By the Cauchy-Schwartz inequality one can show that the last expression is finite. Clearly it does not depend on t

The process $\{X_t\}$ in (1.30) is also called an *infinite order moving average* process, and written as MA(∞)

One way that infinite order moving averages arise is from what is perhaps the most basic time series model, the linear autoregression

In particular, consider the AR(1) model

$$X_t = \phi X_{t-1} + \epsilon_t \quad \text{where} \quad |\phi| < 1 \tag{1.31}$$

By direct substitution, it is easy to verify that the MA(∞) process

$$X_t = \sum_{j=0}^{\infty} \phi^j \epsilon_{t-j} \tag{1.32}$$

is a solution to this model

Comments:

- By a "solution" we mean that X_t is expressed in terms of the parameters and the driving process $\{\epsilon_t\}$
- In fact one can show that (1.32) is the *only* solution to (1.31) that is also covariance stationary

Of course the AR(1) representation (1.31) is still useful, being more parsimonious and often more intuitive

For example, we can use this representation to obtain $\mathbb{E} X_t^2 = (\phi X_{t-1} + \epsilon_t)^2$, and hence, using stationarity, $\mathbb{E} X_{t-1} \epsilon_t = 0$ and $\mathbb{E} \epsilon_t^2 = \sigma^2$,

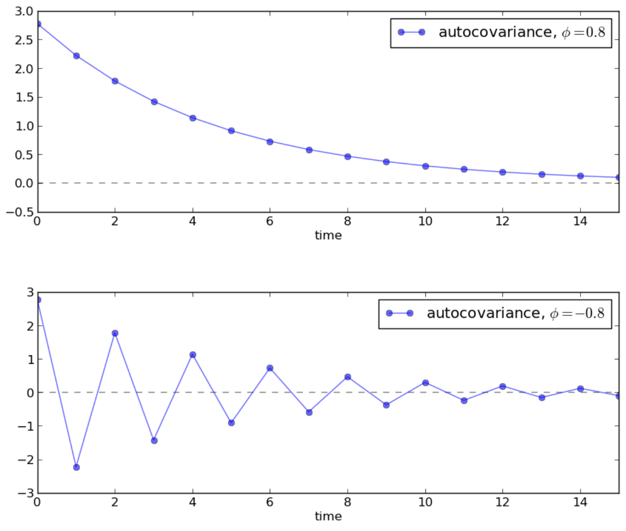
$$\gamma(0) = \phi^2 \gamma(0) + \sigma^2 \tag{1.33}$$

A similar argument can be used to deduce that $\gamma(k) = \phi^k \gamma(k-1)$ for all $k \geq 1$

Combining this difference equation with the initial condition in (1.33), we get the AR(1) autocovariance function

$$\gamma(k) = \phi^k \frac{\sigma^2}{1 - \phi^2}, \qquad k = 0, 1, \ldots \tag{1.34}$$

The next figure plots this function for $\phi = 0.8$ and $\phi = -0.8$



For the MA(1) process $X_t = \epsilon_t + \theta \epsilon_{t-1}$ you will be able to verify that

$$\gamma(0) = \sigma^2 (1 + \theta^2), \quad \gamma(1) = \sigma^2 \theta, \quad \gamma(k) = 0 \quad \forall \, k > 1$$

Spectral Analysis

Autocovariance functions provide a great deal of infomation about covariance stationary processes

In fact, for zero-mean Gaussian processes, the autocovariance function characterizes the entire joint distribution

Even for non-Gaussian processes, it provides a significant amount of information

It turns out that there is an alternative representation of the autocovariance function of a covariance stationary process, called the *spectral density*

At times, the spectral density is easier to derive, easier to manipulate and provides additional intuition

Complex Numbers Before discussing the spectral density, we need to say a word or two about Fourier transforms and complex numbers

For readers who are not confident with complex numbers, it can be helpful to remember that, in a formal sense, complex numbers are just points $(x, y) \in \mathbb{R}^2$ endowed with a specific notion of multiplication

When (x, y) is regarded as a complex number, x is called the *real part* and y is called the *imaginary part*

The *modulus* or *absolute value* of a complex number $z = (x, y)$ is just its Euclidean norm in $\mathbb{R}^2$, but is usually written as $|z|$ instead of $\|z\|$

The product of two complex numbers (x, y) and (u, v) is defined to be $(xu - vy, xv + yu)$, while addition is standard pointwise vector addition

When endowed with these notions of multiplication and addition, the set of complex numbers forms a field — addition and multiplication play well together, just as they do in $\mathbb{R}$

The complex number (x, y) is often written as $x + iy$, where i is called the *imaginary unit*, and is understood to obey $i^2 = -1$

This more common notation can be thought of as a easy way to remember the definition of multiplication given above, because, proceeding naively,

$$(x + iy)(u + iv) = xu - yv + i(xv + yu)$$

Converted back to our first notation, this becomes $(xu - vy, xv + yu)$, which is the same as the product of (x, y) and (u, v) from our previous definition

From now on we use the second notation to represent complex numbers

Complex numbers are sometimes expressed in their polar form $re^{i\omega}$, which should be interpreted as

$$re^{i\omega} := r(\cos(\omega) + i \sin(\omega))$$

Fourier Transforms Let α be a function from $t \in \mathbb{Z}$ to $\alpha(t) \in \mathbb{R}$ (i.e., a real-valued sequence on the integers) satisfying $\sum_{t \in \mathbb{Z}} \alpha(t)^2 < \infty$

The (discrete time) Fourier transform of α is defined to be the complex-valued function

$$\Lambda(\omega) := \sum_{t \in \mathbb{Z}} \alpha(t) e^{-i\omega t}, \qquad \omega \in [0, 2\pi]$$

It's an exercise to show that if α is *even*, in the sense that $\alpha(t) = \alpha(-t)$ for all t, then

$$\Lambda(\omega) = \alpha(0) + 2 \sum_{t \geq 1} \alpha(t) \cos(\omega t)$$

and, in particular, Λ is real-valued

Fourier transforms are important for a variety of reasons, one of them being that the function Λ completely characterizes the sequence α

In fact the latter can be easily recovered from the former, as we'll see below

Spectral Densities Let $\{X_t\}$ be a covariance stationary process with autocovariance function γ satisfying $\sum_{t \in \mathbb{Z}} \gamma(t)^2 < \infty$

The *spectral density* f of $\{X_t\}$ is defined as the Fourier transform of its autocovariance function γ

That is,

$$f(\omega) = \sum_{t \in \mathbb{Z}} \gamma(t) e^{-i\omega t}, \qquad \omega \in \mathbb{R}$$

(Some authors normalize the expression on the right by $1/\pi$ or $1/2\pi$ — the chosen convention makes little difference provided you are consistent)

It is not difficult to confirm that f is even, and also 2π-periodic, in the sense that $f(2\pi + \omega) = f(\omega)$ for all ω

It follows that the values of f on $[0, \pi]$ determine the values of f on $[0, 2\pi]$, because if $x \in [0, \pi]$, then

$$f(\pi + x) = f(-\pi - x) = f(-\pi - x + 2\pi) = f(\pi - x)$$

In fact, since f is 2π-periodic, this means that the values of f on $[0, \pi]$ determine its values everywhere on $\mathbb{R}$

For this reason it is standard to plot the spectral density only on the interval $[0, \pi]$

Examples Consider a white noise process $\{\epsilon_t\}$ with standard deviation σ

In this case we have

$$f(\omega) = \gamma(0) e^{-i\omega 0} = \sigma^2$$

As we will see, the fact that f is constant in this case can be interpreted as meaning that "all frequencies are equally present"

(White light has this property when frequency refers to the visible spectrum, a connection that provides the origins of the term "white noise")

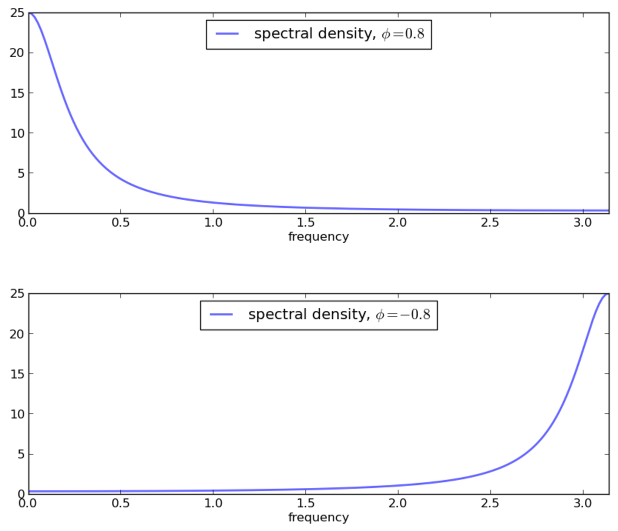
It is an exercise to show that the MA(1) process $X_t = \theta X_{t-1} + \epsilon_t$ has spectral density

$$f(\omega) = \sigma^2(1 + 2\theta\cos(\omega) + \theta^2)$$

With a bit more effort, it's possible to show (see, e.g., p. 261 of [Sargent1987]) that the spectral density of the AR(1) process $X_t = \phi X_{t-1} + \epsilon_t$ is

$$f(\omega) = \frac{\sigma^2}{1 - 2\phi\cos(\omega) + \phi^2} \tag{1.35}$$

Interpreting the Spectral Density Plotting (1.35) reveals the shape of the spectral density for the AR(1) model when ϕ takes the values 0.8 and -0.8 respectively



These spectral densities correspond to the autocovariance functions for the AR(1) process *shown above*

Informally, we think of the spectral density as being large at those $\omega \in [0, \pi]$ such that the autocovariance function exhibits significant cycles at this "frequency"

To see the idea, let's consider why, in the lower panel of the preceding figure, the spectral density for the case $\phi = -0.8$ is large at $\omega = \pi$

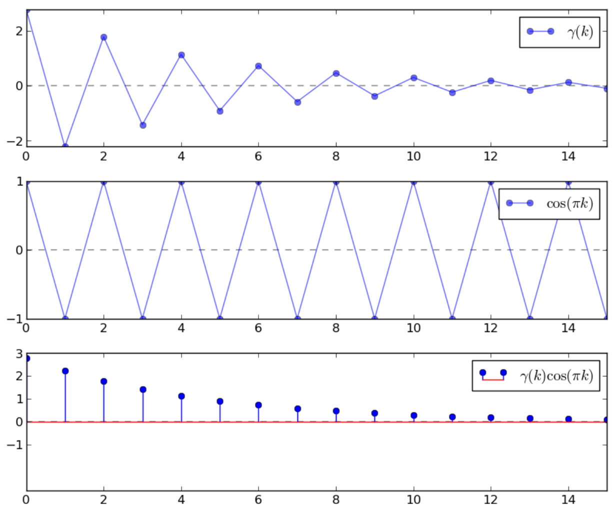
Recall that the spectral density can be expressed as

$$f(\omega) = \gamma(0) + 2\sum_{k\geq 1}\gamma(k)\cos(\omega k) = \gamma(0) + 2\sum_{k\geq 1}(-0.8)^k\cos(\omega k) \tag{1.36}$$

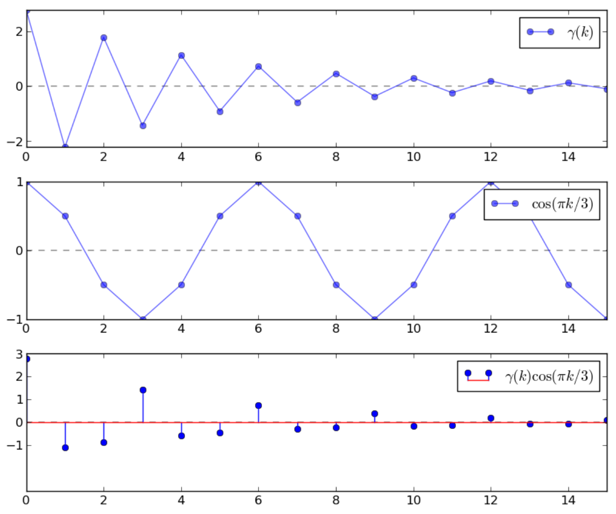
When we evaluate this at $\omega = \pi$, we get a large number because $\cos(\pi k)$ is large and positive when $(-0.8)^k$ is positive, and large and negative when $(-0.8)^k$ is negative

Hence the product is always large and positive, and hence the sum of the products on the right-hand side of (1.36) is large

These ideas are illustrated in the next figure, which has k on the horizontal axis (click to enlarge)



On the other hand, if we evaluate $f(\omega)$ at $\omega = \pi/3$, then the cycles are not matched, the sequence $\gamma(k)\cos(\omega k)$ contains both positive and negative terms, and hence the sum of these terms is much smaller



In summary, the spectral density is large at frequencies ω where the autocovariance function exhibits cycles

Inverting the Transformation There is a clear sense in which the mapping from autocovariance functions to spectral densities is a bijection — for details see the Riesz–Fischer theorem

In particular, we can always recover the autocovariance function from the spectral density via the *inverse Fourier* *transform*

$$\gamma(k) = \frac{1}{2\pi} \int_0^{2\pi} f(\omega) e^{i\omega k}, \qquad k \in \mathbb{Z} \tag{1.37}$$

This is convenient because it is often easier to calculate spectral densities and then use this transform than to compute autocovariance functions directly from the definition

ARMA Processes

An important class of linear processes are the so-called *autoregressive moving average (ARMA) processes*, a typical instance of which is

$$X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q} \tag{1.38}$$

This particular process is called ARMA(p, q) to indicate the number of autoregressive and moving average parameters

There is an alternative notation in common use, based around the *lag operator* L, the action of which is defined for arbitrary variable Y_t as $L^k Y_t = Y_{t-k}$

With this notation we can rewrite (1.38) as

$$L^0 X_t - \phi_1 L^1 X_t - \cdots - \phi_p L^p X_t = L^0 \epsilon_t + \theta_1 L^1 \epsilon_t + \cdots + \theta_q L^q \epsilon_t \tag{1.39}$$

If we let $\phi(z)$ and $\theta(z)$ be the polynomials

$$\phi(z) := 1 - \phi_1 z - \cdots - \phi_p z^p \quad \text{and} \quad \theta(z) := 1 + \theta_1 z + \cdots + \theta_q z^q$$

then (1.39) simplifies to

$$\phi(L) X_t = \theta(L) \epsilon_t \tag{1.40}$$

This is a very succinct expression

On top of that, it turns out that algebraic manipulations of the lag operator — treating it as an ordinary scalar — often lead to correct solutions

In what follows we always assume that the roots of the polynomial $\phi(z)$ lie outside the unit circle in the complex plane

This condition is sufficient to guarantee covariance stationarity

ARMAs as MAs The set of ARMA(p, q) processes falls within the class of generalized linear processes *described above*

In particular, given the parameters of a covariance stationary ARMA(p, q) process, there exists a sequence of weights $\{\psi_t\}$ with $\sum_{t=0}^{\infty} \psi_t^2 < \infty$ and $X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}$ for all t

The sequence $\{\psi_t\}$ can be obtained by a recursive procedure outlined on page 79 of [CryerChan2008]

In the present context, the function $t \mapsto \psi_t$ is often called the *impulse response function*

The ARMA Spectral Density From the impulse response function, one can compute the autocovariance function — see page 85 of [CryerChan2008] for details

However, the expression is a little awkward, and, as a result, it's more common to calculate the spectral density and then compute the autocovariance function where necessary using the inverse Fourier transform (1.37)

The spectral density of the ARMA process (1.38) is given by

$$f(\omega) = \left| \frac{\theta(e^{i\omega})}{\phi(e^{i\omega})} \right|^2 \sigma^2 \tag{1.41}$$

Here σ is the standard deviation of the white noise process $\{\epsilon_t\}$

The derivation of (1.41) is based around the fact that convolutions become products under Fourier transformations — see, for example, [Sargent1987], chapter 11, section 4

Implementation

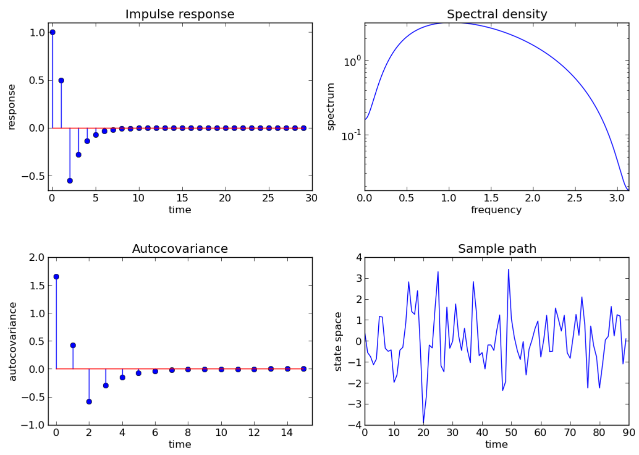
In this lecture, our main objective is to provide Python code to represent ARMA(p, q) processes visually via their

1. impulse response function
2. simulated time series
3. autocovariance function
4. spectral density

In additional to individual plots of these entities, we want to provide functionality to generate 2x2 plots containing all this information

In other words, we want to replicate the plots on pages 68–69 of RMT3

Here's an example corresponding to the model $X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2}$



Before presenting the code, we make some brief comments on the implementation

Comments on the Structure of the Program To achieve our stated aims we will implement a class called `linearProcess` with methods that generate the desired plots

The call

```
lp = linearProcess(phi, theta, sigma)
```

will create an instance `lp` that represents the ARMA(p, q) model

$$X_t = \phi_1 X_{t-1} + ... + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q}$$

If `phi` and `theta` are arrays or sequences, then the interpretation will be

- `phi` holds the vector of parameters $(\phi_1, \phi_2, ..., \phi_p)$
- `theta` holds the vector of parameters $(\theta_1, \theta_2, ..., \theta_q)$

The parameter `sigma` is always a scalar, the standard deviation of the white noise

We will also permit `phi` and `theta` to be scalars, in which case the model will be interpreted as

$$X_t = \phi X_{t-1} + \epsilon_t + \theta \epsilon_{t-1}$$

The two packages most useful for working with ARMA models are `scipy.signal` and `numpy.fft`

The package `scipy.signal` expects the parameters to be passed in to its functions in a manner consistent with the alternative ARMA notation (1.40)

For example, the impulse response sequence $\{\psi_t\}$ discussed above can be obtained using `scipy.signal.dimpulse`, and the function call should be of the form

```
times, psi = dimpulse((num, den, 1), n=impulse_length)
```

where

- `num` is the vector $(1, \theta_1, \theta_2, \ldots, \theta_q)$
- `den` is the vector $(1, -\phi_1, -\phi_2, \ldots, -\phi_p)$

To this end, we will also maintain the arrays `num` and `den` as instance data, with their values computed automatically from the values of `phi` and `theta` supplied by the user

If the user decides to change the value of either `phi` or `theta` ex-post by assignments such as

```
lp.phi = (0.5, 0.2)
lp.theta = (0, -0.1)
```

then we would like `num` and `den` to update automatically to reflect these new parameters

This will be achieved in our implementation by using *Descriptors*

Computing the Autocovariance Function As discussed above, for ARMA processes the spectral density has a *simple representation* that is relatively easy to calculate

Given this fact, the easiest way to obtain the autocovariance function is to recover it from the spectral density via the inverse Fourier transform

Here we will use NumPy's Fourier transform packaget *np.fft*, which wraps a standard Fortran-based package called FFTPACK

A look at the np.fft documentation shows that the inverse transform *np.fft.ifft* takes a given sequence $A_0, A_1, \dots, A_{n-1}$ and returns the sequence $a_0, a_1, \dots, a_{n-1}$ defined by

$$a_k = \frac{1}{n} \sum_{t=0}^{n-1} A_t e^{ik2\pi t/n}$$

Thus, if we set $A_t = f(\omega_t)$, where f is the spectral density and $\omega_t := 2\pi t/n$, then

$$a_k = \frac{1}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i\omega_t k} = \frac{1}{2\pi} \frac{2\pi}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i\omega_t k}, \qquad \omega_t := 2\pi t/n$$

For n sufficiently large, we then have

$$a_k \approx \frac{1}{2\pi} \int_0^{2\pi} f(\omega) e^{i\omega k} d\omega$$

In view of (1.37) we have now shown that, for n sufficiently large, $a_k \approx \gamma(k)$ — which is exactly what we want to compute

Code Our implementation is as follows

```
import numpy as np
from numpy import conj, pi, real
import matplotlib.pyplot as plt
from scipy.signal import dimpulse, freqz, dlsim


class linearProcess(object):
    """
    This class provides functions for working with scalar ARMA processes.  In
    particular, it defines methods for computing and plotting the
    autocovariance function, the spectral density, the impulse-response
    function and simulated time series.

    """

    def __init__(self, phi, theta=0, sigma=1) :
        """
        This class represents scalar ARMA(p, q) processes.  The parameters phi
        and theta can be NumPy arrays, array-like sequences (lists, tuples) or
        scalars.

        If phi and theta are scalars, then the model is
        understood to be

            X_t = phi X_{t-1} + epsilon_t + theta epsilon_{t-1}

        where {epsilon_t} is a white noise process with standard deviation
        sigma.  If phi and theta are arrays or sequences, then the
        interpretation is the ARMA(p, q) model

            X_t = phi_1 X_{t-1} + ... + phi_p X_{t-p} +
                epsilon_t + theta_1 epsilon_{t-1} + ... + theta_q epsilon_{t-q}

        where

            * phi = (phi_1, phi_2,..., phi_p)
            * theta = (theta_1, theta_2,..., theta_q)
            * sigma is a scalar, the standard deviation of the white noise

        """
        self._phi, self._theta = phi, theta
        self.sigma = sigma
        self.set_params()

    def get_phi(self):
        return self._phi

    def get_theta(self):
        return self._theta

    def set_phi(self, new_value):
        self._phi = new_value
        self.set_params()

    def set_theta(self, new_value):
        self._theta = new_value
        self.set_params()

    phi = property(get_phi, set_phi)
    theta = property(get_theta, set_theta)

    def set_params(self):
        """
        Internally, scipy.signal works with systems of the form

            den(L) X_t = num(L) epsilon_t

        where L is the lag operator. To match this, we set

            den = (1, -phi_1, -phi_2,..., -phi_p)
            num = (1, theta_1, theta_2,..., theta_q)

        In addition, den must be at least as long as num.  This can be
        achieved by padding it out with zeros when required.
        """
        num = np.asarray(self._theta)
        if np.isscalar(self._phi):
            den = np.array(-self._phi)
        else:
            den = -np.asarray(self._phi)
        self.num = np.insert(num, 0, 1)       # The array (1, theta)
        self.den = np.insert(den, 0, 1)       # The array (1, -phi)
        if len(self.den) < len(self.num):     # Pad den with zeros if necessary
            temp = np.zeros(len(self.num) - len(self.den))
            self.den = np.hstack((self.den, temp))

    def impulse_response(self, impulse_length=30):
        """
        Get the impulse response corresponding to our model.  Returns psi,
        where psi[j] is the response at lag j.  Note: psi[0] is unity.
        """
        sys = self.num, self.den, 1
        times, psi = dimpulse(sys, n=impulse_length)
        psi = psi[0].flatten()  # Simplify return value into flat array
        return psi

    def spectral_density(self, domain_max=2*pi, resolution=1e5) :
        """
        Compute the spectral density function over domain [0, domain_max].
        The spectral density is the discrete time Fourier transform of the
        autocovariance function.  In particular,

            f(w) = sum_k gamma(k) exp(-ikw)

        where gamma is the autocovariance function and the sum is over k in Z,
        the set of all integers.
        """
        w = np.linspace(0, domain_max, resolution)
        h = freqz(self.num, self.den, w)[1]
        spect = h * conj(h) * self.sigma**2
        return w, spect

    def autocovariance(self, num_autocov=16) :
        """
        Compute the autocovariance function over the integers
        range(num_autocov) using the spectral density and the inverse Fourier
        transform.
        """
        spect = self.spectral_density()[1]
        acov = np.fft.ifft(spect).real
        return acov[:num_autocov]  # num_autocov should be <= len(acov) / 2

    def simulation(self, ts_length=90) :
        " Compute a simulated sample path. "
        sys = self.num, self.den, 1
        u = np.random.randn(ts_length, 1)
        return dlsim(sys, u)[1]

    def plot_impulse_response(self, ax=None, show=True):
        if show:
            fig, ax = plt.subplots()
        ax.set_title('Impulse response')
        yi = self.impulse_response()
        ax.stem(range(len(yi)), yi)
        ax.set_xlim(xmin=(-0.5))
        ax.set_ylim(min(yi)-0.1,max(yi)+0.1)
        ax.set_xlabel('time')
        ax.set_ylabel('response')
        if show:
            fig.show()

    def plot_spectral_density(self, ax=None, show=True):
        if show:
            fig, ax = plt.subplots()
        ax.set_title('Spectral density')
        w, spect = self.spectral_density(domain_max=pi)
        ax.semilogy(w, spect)
        ax.set_xlim(0, pi)
        ax.set_ylim(0, np.max(spect))
        ax.set_xlabel('frequency')
        ax.set_ylabel('spectrum')
        if show:
            fig.show()

    def plot_autocovariance(self, ax=None, show=True):
        if show:
            fig, ax = plt.subplots()
        ax.set_title('Autocovariance')
        acov = self.autocovariance()
        ax.stem(range(len(acov)), acov)
        ax.set_xlim(-0.5, len(acov) - 0.5)
        ax.set_xlabel('time')
        ax.set_ylabel('autocovariance')
        if show:
            fig.show()

    def plot_simulation(self, ax=None, show=True):
        if show:
            fig, ax = plt.subplots()
        ax.set_title('Sample path')
        x_out = self.simulation()
        ax.plot(x_out)
        ax.set_xlabel('time')
        ax.set_ylabel('state space')
        if show:
            fig.show()

    def quad_plot(self) :
        """
        Plots the impulse response, spectral_density, autocovariance, and one
        realization of the process.
        """
        num_rows, num_cols = 2, 2
        fig, axes = plt.subplots(num_rows, num_cols, figsize=(12, 8))
        plt.subplots_adjust(hspace=0.4)
        self.plot_impulse_response(axes[0, 0], show=False)
        self.plot_spectral_density(axes[0, 1], show=False)
        self.plot_autocovariance(axes[1, 0], show=False)
        self.plot_simulation(axes[1, 1], show=False)
        fig.show()
```

As an example of usage, try

```
phi = 0.5
theta = 0, -0.8
lp = linearProcess(phi, theta)
lp.quad_plot()
```

Exercises

To be written. How about an exercise to add functionality that inspects the parameters and verifies stationarity (by checking that the relevant roots are outside the unit circle in C)?

References

On-the-Job Search

Overview

In this section we solve a simple on-the-job search model

- based on RMT3, exercise 6.18
- see also [add Jovanovic reference]

Model features

- job-specific human capital accumulation combined with on-the-job search
- infinite horizon dynamic programming with one state variable and two controls

Model

Let

- x_t denote the time-t job-specific human capital of a worker employed at a given firm
- w_t denote current wages

Let $w_t = x_t(1 - s_t - \phi_t)$, where

- ϕ_t is investment in job-specific human capital for the current role
- s_t is search effort, devoted to obtaining new offers from other firms.

For as long as the worker remains in the current job, evolution of $\{x_t\}$ is given by $x_{t+1} = G(x_t, \phi_t)$

When search effort at t is s_t, the worker receives a new job offer with probability $\pi(s_t) \in [0, 1]$

Value of offer is U_{t+1}, where $\{U_t\}$ is iid with common distribution F

Worker has the right to reject the current offer and continue with existing job.

In particular, $x_{t+1} = U_{t+1}$ if accepts and $x_{t+1} = G(x_t, \phi_t)$ if rejects

Letting $b_{t+1} \in \{0, 1\}$ be binary with $b_{t+1} = 1$ indicating an offer, we can write

$$x_{t+1} = (1 - b_{t+1}) G(x_t, \phi_t) + b_{t+1} \max\{G(x_t, \phi_t), U_{t+1}\} \tag{1.42}$$

Agent's objective: maximize expected discounted sum of wages via controls $\{s_t\}$ and $\{\phi_t\}$

Taking the expectation of $V(x_{t+1})$ and using (1.42), the Bellman equation for this problem can be written as

$$V(x) = \max_{s+\phi \leq 1} \left\{ x(1 - s - \phi) + \beta(1 - \pi(s)) V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \vee u] F(du) \right\}. \tag{1.43}$$

Here nonnegativity of s and ϕ is understood, while $a \vee b := \max\{a, b\}$

Parameterization In the implementation below, we will focus on the parameterization

$$G(x,\phi) = A(x\phi)^{\alpha}, \quad \pi(s) = \sqrt{s} \quad \text{and} \quad F = \text{Beta}(2,2)$$

with default parameter values

- $A = 1.4$
- $\alpha = 0.6$
- $\beta = 0.96$

The Beta(2,2) distribution is supported on $(0, 1)$. It has a unimodal, symmetric density peaked at 0.5.

Back-of-the-Envelope Calculations Before we solve the model, let's make some quick calculations that provide intuition on what the solution should look like.

To begin, observe that the worker has two instruments to build capital and hence wages:

1. invest in capital specific to the current job via ϕ
2. search for a new job with better job-specific capital match via s

Since wages are $x(1 - s - \phi)$, marginal cost of investment via either ϕ or s is identical

Our risk neutral worker should focus on whatever instrument has the highest expected return

The relative expected return will depend on x

For example, suppose first that $x = 0.05$

- If $s = 1$ and $\phi = 0$, then since $G(x, \phi) = 0$, taking expectations of (1.42) gives expected next period capital equal to $\pi(s)\mathbb{E}U = \mathbb{E}U = 0.5$
- If $s = 0$ and $\phi = 1$, then next period capital is $G(x, \phi) = G(0.05, 1) \approx 0.23$

Both rates of return are good, but the return from search is better

Next suppose that $x = 0.4$

- If $s = 1$ and $\phi = 0$, then expected next period capital is again 0.5
- If $s = 0$ and $\phi = 1$, then $G(x, \phi) = G(0.4, 1) \approx 0.8$

Regturn from investment via ϕ dominates expected return from search

Combining these observations give us two informal predictions:

1. At any given state x, the two controls ϕ and s will function primarily as substitutes — worker will focus on whichever instrument has the higher expected return
2. For sufficiently small x, search will be preferable to investment in job-specific human capital. For larger x, the reverse will be true

Now let's turn to implementation, and see if we can match our predictions.

Implementation: `jv.py`

This section describes the module `jv`, which solves the DP problem described above

As with all other code, the file can be found in the main repository

Code The code listing is followed by a detailed explanation — skim the code and move on to the explanation before reading the code in detail

```
import numpy as np
from scipy.integrate import fixed_quad as integrate
from scipy.optimize import fmin_slsqp as minimize
import scipy.stats as stats
from scipy import interp

epsilon = 1e-4  #  A small number, used in the optimization routine

class workerProblem:

    def __init__(self, A=1.4, alpha=0.6, beta=0.96, grid_size=50):
        """
        This class is just a "struct" to hold the attributes of a given model.
        """
        self.A, self.alpha, self.beta = A, alpha, beta
        # Set defaults for G, pi and F
        self.G = lambda x, phi: A * (x * phi)**alpha
        self.pi = np.sqrt
        self.F = stats.beta(2, 2)
        # Set up grid over the state space for DP.  Max of grid is the max of
        # a large quantile value for F and the fixed point y = G(y, 1).
        grid_max = max(A**(1 / (1 - alpha)), self.F.ppf(1 - epsilon))
        self.x_grid = np.linspace(epsilon, grid_max, grid_size)


def bellman_operator(wp, V, brute_force=False, return_policies=False):
    """
    Parameter wp is an instance of workerProblem.  Thus function returns the
    approximate value function TV by applying the Bellman operator associated
    with the model wp to the function V.  Returns TV, or the V-greedy policies
    s_policy and phi_policy when return_policies=True.

    In the function, the array V is replaced below with a function Vf that
    implements linear interpolation over the points (V(x), x) for x in x_grid.
    If the brute_force flag is true, then grid search is performed at each
    maximization step.  In either case, T returns a NumPy array representing
    the updated values TV(x) over x in x_grid.

    """
    G, pi, F, beta = wp.G, wp.pi, wp.F, wp.beta  # Simplify names
    Vf = lambda x: interp(x, wp.x_grid, V)
    N = len(wp.x_grid)
    new_V, s_policy, phi_policy = np.empty(N), np.empty(N), np.empty(N)
    a, b = F.ppf(0.005), F.ppf(0.995)   # Quantiles, for integration
    c1 = lambda z: 1 - sum(z)           # used to enforce s + phi <= 1
    c2 = lambda z: z[0] - epsilon       # used to enforce s >= epsilon
    c3 = lambda z: z[1] - epsilon       # used to enforce phi >= epsilon
    guess, constraints = (0.2, 0.2), [c1, c2, c3]

    for i, x in enumerate(wp.x_grid):

        def w(z):  # z = (s, phi)
            """
            Objective function, corresponding to the right-hand side of the
            Bellman equation. Negated because we will minimize.
            """
            s, phi = z
            integrand = lambda u: Vf(np.maximum(G(x, phi), u)) * F.pdf(u)
            integral, err = integrate(integrand, a, b)
            q = pi(s) * integral + (1 - pi(s)) * Vf(G(x, phi))
            return - x * (1 - phi - s) - beta * q

        if not brute_force:  # Use SciPy solver
            max_s, max_phi = minimize(w, guess, ieqcons=constraints, disp=0)
            max_val = -w((max_s, max_phi))

        else:  # Search on a grid
            search_grid = np.linspace(epsilon, 1, 15)
            max_val = -1
            for s in search_grid:
                for phi in search_grid:
                    current_val = -w((s, phi)) if s + phi <= 1 else -1
                    if current_val > max_val:
                        max_val, max_s, max_phi = current_val, s, phi

        new_V[i] = max_val
        s_policy[i], phi_policy[i] = max_s, max_phi

    if return_policies:
        return s_policy, phi_policy
    else:
        return new_V
```

Explanation The file `jv.py` begins with a few quick imports

- `fixed_quad` is a simple non-adaptive integration routine
- `fmin_slsqp` is a minimization routine that permits inequality constraints

Next we build a simple class called `workerProblem` that packages all the parameters and other basic attributes of a given model

What's the point of defining such a class?

The point is that when we come to writing the code for the Bellman operator, we want to make it relatively generic—and hence reusable.

- For example, use generic $G(x, \phi)$ instead of specific $A(x\phi)^\alpha$.

So any specifics need to be passed in to the Bellman operator when we call it.

To avoid a long list of parameters, it is convenient to wrap all the specifics in a single class, an instance of which can be passed to the operator—hence the class `workerProblem`

Regarding the function `bellman_operator()`, it takes as arguments

- an instance of `workerProblem`, which contains all the specifics of a particular parameterization
- a candidate value function V to be updated to TV via

$$TV(x) = - \min_{s+\phi \leq 1} w(s, \phi)$$

where

$$w(s, \phi) := -\left\{x(1 - s - \phi) + \beta(1 - \pi(s))V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \vee u]F(du)\right\} \tag{1.44}$$

Here we are minimizing instead of maximizing to fit with SciPy's optimization routines

When we represent V, it will be with a NumPy array V giving values on grid x_grid

But to evaluate the right-hand side of (1.44), we need a function, so we replace the arrays V and x_grid with a function Vf that gives linear iterpolation of V on x_grid

In the preliminaries of the function bellman_operator()

- from the array V we define a linear interpolation Vf of its values
- c1 is used to implement the constraint $s + \phi \leq 1$
- c2 is used to implement $s \geq \epsilon$, a numerically stable alternative to the true constraint $s \geq 0$
- c3 does the same for ϕ

Inside the for loop, for each x in the grid over the state space, we set up the function $w(z) = w(s, \phi)$ defined in (1.44).

The function is minimized over all feasible (s, ϕ) pairs, either by

- a relatively sophisticated solver from SciPy called fmin_slsqp, or
- brute force search over a grid

The former is much faster, but convergence to the global optimum is not guaranteed. Grid search is a simple way to check results

Solving for Policies

Let's plot the optimal policies and see what they look like

The code is in a file jv_test.py that imports jv (available from the main repository) and looks as follows

```python
import matplotlib.pyplot as plt
from jv import workerProblem, bellman_operator
from compute_fp import compute_fixed_point

wp = workerProblem(grid_size=25)

v_init = wp.x_grid * 0.5
V = compute_fixed_point(bellman_operator, wp, v_init, max_iter=40)
s_policy, phi_policy = bellman_operator(wp, V, return_policies=True)

fig, ax = plt.subplots()
ax.set_xlim(0, max(wp.x_grid))
ax.set_ylim(-0.1, 1.1)
ax.plot(wp.x_grid, phi_policy, 'b-', label='phi')
ax.plot(wp.x_grid, s_policy, 'g-', label='s')
ax.legend()
plt.show()
```

It produces the following figure

The horizontal axis is the state x, while the vertical axis gives $s(x)$ and $\phi(x)$

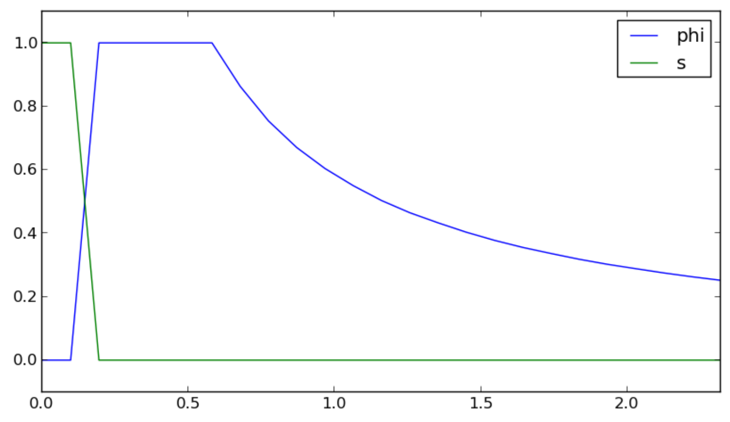


Figure 1.3: Optimal policies

Overall, the policies match well with our predictions from section *Back-of-the-Envelope Calculations*.

- Worker switches from one investment strategy to the other depending on relative return
- For low values of x, the best option is to search for a new job
- Once x is larger, worker does better by investing in human capital specific to the current position

Exercises

Exercise 1 Let's look at the dynamics for the state process $\{x_t\}$ associated with these policies.

The dynamics are given by (1.42) when ϕ_t and s_t are chosen according to the optimal policies, and $\mathbb{P}\{b_{t+1} = 1\} = \pi(s_t)$.

Since the dynamics are random, analysis is a bit subtle

One way to do it is to plot, for each x in a relatively fine grid called `plot_grid`, a large number K of realizations of x_{t+1} given $x_t = x$. Plot this with one dot for each realization, in the form of a 45 degree diagram. Set:

```
K = 50
plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
fig, ax = plt.subplots()
ax.set_xlim(0, plot_grid_max)
ax.set_ylim(0, plot_grid_max)
```

By examining the plot, argue that under the optimal policies, the state x_t will converge to a constant value $\bar{x}$ close to unity

Argue that at the steady state, $s_t \approx 0$ and $\phi_t \approx 0.6$.

Solution: *View solution*

Exercise 2 In the preceding exercise we found that s_t converges to zero and ϕ_t converges to about 0.6

Since these results were calculated at a value of β close to one, let's compare them to the best choice for an *infinitely* patient worker.

Intuitively, an infinitely patient worker would like to maximize steady state wages, which are a function of steady state capital.

You can take it as given—it's certainly true—that the infinitely patient worker does not search in the long run (i.e., $s_t = 0$ for large t)

Thus, given ϕ, steady state capital is the positive fixed point $x^*(\phi)$ of the map $x \mapsto G(x, \phi)$.

Steady state wages can be written as $w^*(\phi) = x^*(\phi)(1 - \phi)$

Graph $w^*(\phi)$ with respect to ϕ, and examine the best choice of ϕ

Can you give a rough intepretation for the value that you see?

Solution: *View solution*

Search with Offer Distribution Unknown

Overview

In this lecture we consider an extension of the job search model developed by John J. McCall [McCall1970]

In the McCall model, an unemployed worker decides when to accept a permanent position at a specified wage, given

- his or her discount rate
- the level of unemployment compensation
- the distribution from which wage offers are drawn

In the version considered below, the wage distribution is unknown and must be learned

- Based on the presentation in RMT3, section 6.6

Model features

- Infinite horizon dynamic programming with two states and one binary control
- Bayesian updating to learn the unknown distribution

Model

Let's first recall the basic McCall model [McCall1970] and then add the variation we want to consider

The Basic McCall Model Consider an unemployed worker who is presented in each period with a permanent job offer at wage w_t

At time t, our worker has two choices

1. Accept the offer and work permanently at constant wage w_t
2. Reject the offer, receive unemployment compensation c, and reconsider next period

The wage sequence $\{w_t\}$ is iid and generated from known density h

The worker aims to maximize the expected discounted sum of earnings $\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$

Trade-off:

- Waiting too long for a good offer is costly, since the future is discounted
- Accepting too early is costly, since better offers will arrive with probability one

Let $V(w)$ denote the maximal expected discounted sum of earnings that can be obtained by an unemployed worker who starts with wage offer w in hand

The function V satisfies the recursion

$$V(w) = \max \left\{ \frac{w}{1-\beta}, \, c + \beta \int V(w')h(w')dw' \right\} \tag{1.45}$$

where the two terms on the r.h.s. are the respective payoffs from accepting and rejecting the current offer w

The optimal policy is a map from states into actions, and hence a binary function of w

Not surprisingly, it turns out to have the form $\mathbb{1}\{w \geq \bar{w}\}$, where

- $\bar{w}$ is a constant depending on (β, h, c) called the *reservation wage*
- $\mathbb{1}\{w \geq \bar{w}\}$ is an indicator function returning 1 if $w \geq \bar{w}$ and 0 otherwise
- 1 indicates "accept" and 0 indicates "reject"

For further details see RMT3, section 6.3

Offer Distribution Unknown Now let's extend the model by considering the variation presented in RMT3, section 6.6

The model is as above, apart from the fact that

- the density h is unknown
- the worker learns about h by starting with a prior and updating based on wage offers that he/she observes

The worker knows there are two possible distributions F and G — with densities f and g

At the start of time, "nature" selects h to be either f or g — the wage distribution from which the entire sequence $\{w_t\}$ will be drawn

This choice is not observed by the worker, who puts prior probability π_0 on f being chosen

Update rule: worker's time t estimate of the distribution is $\pi_t f + (1-\pi_t)g$, where π_t updates via

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1-\pi_t)g(w_{t+1})} \tag{1.46}$$

This last expression follows from Bayes' rule, which tells us that

$$\mathbb{P}\{h = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, h = f\}\mathbb{P}\{h = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\psi \in \{f,g\}} \mathbb{P}\{W = w \,|\, h = \psi\}\mathbb{P}\{h = \psi\}$$

The fact that (1.46) is recursive allows us to progress to a recursive solution method

Letting

$$h_\pi(w) := \pi f(w) + (1-\pi)g(w) \quad \text{and} \quad q(w, \pi) := \frac{\pi f(w)}{\pi f(w) + (1-\pi)g(w)}$$

we can express the value function for the unemployed worker recursively as follows

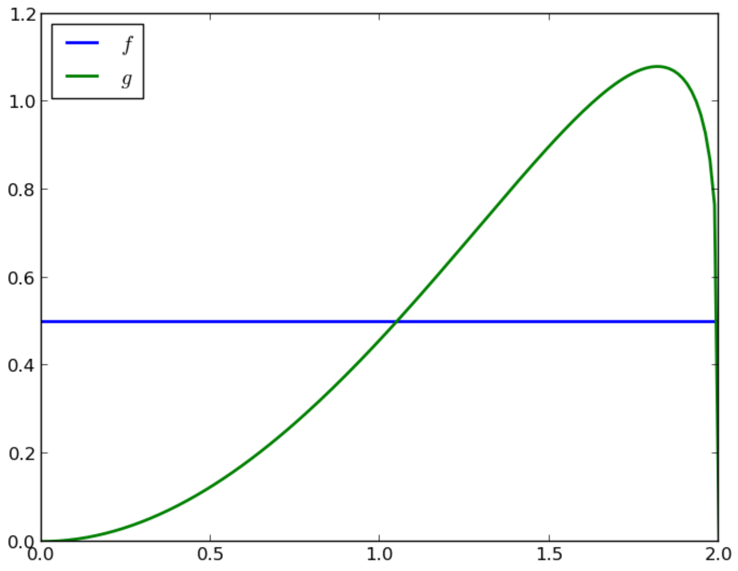
$$V(w, \pi) = \max \left\{ \frac{w}{1-\beta}, \, c + \beta \int V(w', \pi') \, h_\pi(w') \, dw' \right\} \quad \text{where} \quad \pi' = q(w', \pi) \tag{1.47}$$

Notice that the current guess π is a state variable, since it affects the worker's perception of probabilities for future rewards

Parameterization Following section 6.6 of RMT3, our baseline parameterization will be

- $f = \text{Beta}(1, 1)$ and $g = \text{Beta}(3, 1.2)$
- $\beta = 0.95$ and $c = 0.6$

The densities f and g have the following shape



Looking Forward What kind of optimal policy might result from (1.47) and the parameterization specified above?

Intuitively, if we accept at w_a and $w_a \leq w_b$, then — all other things being given — we should also accept at w_b

This suggests a policy of accepting whenever w exceeds some threshold value $\bar{w}$

But $\bar{w}$ should depend on π — in fact it should be decreasing in π because

- f is a less attractive offer distribution than g
- larger π means more weight on f and less on g

Thus larger π depresses the worker's assessment of her future prospects, and relatively low current offers become more attractive

Summary: We conjecture that that the optimal policy is of the form $\mathbb{1}\{w \geq \bar{w}(\pi)\}$ for some decreasing function $\bar{w}$

Take 1: Solution by VFI

Let's set about solving the model and see how our results match with our intuition

We begin by solving via value function iteration (VFI), which is natural but ultimately turns out to be second best

VFI is implemented in the module `odu_vfi`, provided in the main repository

The code is as follows — but read discussion given beneath it first

```
"""
Origin: QEwP by John Stachurski and Thomas J. Sargent
Date:   3/2013
File:   odu_vfi.py

Solves the "Offer Distribution Unknown" Model by value function iteration.

Note that a much better technique is given in solution_odu_ex1.py
"""
from scipy.interpolate import LinearNDInterpolator
from scipy.integrate import fixed_quad
from scipy.stats import beta as beta_distribution
import numpy as np

class searchProblem:
    """
    A class to store a given parameterization of the "offer distribution
    unknown" model.
    """

    def __init__(self, beta=0.95, c=0.6, F_a=1, F_b=1, G_a=3, G_b=1.2,
            w_max=2, w_grid_size=40, pi_grid_size=40):
        """
        Sets up parameters and grid.  The attribute "grid_points" defined
        below is a 2 column array that stores the 2D grid points for the DP
        problem. Each row represents a single (w, pi) pair.
        """
        self.beta, self.c, self.w_max = beta, c, w_max
        self.F = beta_distribution(F_a, F_b, scale=w_max)
        self.G = beta_distribution(G_a, G_b, scale=w_max)
        self.f, self.g = self.F.pdf, self.G.pdf    # Density functions
        self.pi_min, self.pi_max = 1e-3, 1 - 1e-3  # Avoids instability
        self.w_grid = np.linspace(0, w_max, w_grid_size)
        self.pi_grid = np.linspace(self.pi_min, self.pi_max, pi_grid_size)
        x, y = np.meshgrid(self.w_grid, self.pi_grid)
        self.grid_points = np.column_stack((x.ravel(1), y.ravel(1)))

    def q(self, w, pi):
        """
        Updates pi using Bayes' rule and the current wage observation w.
        """
        new_pi = 1.0 / (1 + ((1 - pi) * self.g(w)) / (pi * self.f(w)))
        # Return new_pi when in [pi_min, pi_max], and the end points otherwise
        return np.maximum(np.minimum(new_pi, self.pi_max), self.pi_min)


def bellman(sp, v):
    """
    The Bellman operator.
```

```
        * sp is an instance of searchProblem
        * v is an approximate value function represented as a one-dimensional
          array.
    """
    f, g, beta, c, q = sp.f, sp.g, sp.beta, sp.c, sp.q  # Simplify names
    vf = LinearNDInterpolator(sp.grid_points, v)
    N = len(v)
    new_v = np.empty(N)
    for i in range(N):
        w, pi = sp.grid_points[i,:]
        v1 = w / (1 - beta)
        integrand = lambda m: vf(m, q(m, pi)) * (pi * f(m) + (1 - pi) * g(m))
        integral, error = fixed_quad(integrand, 0, sp.w_max)
        v2 = c + beta * integral
        new_v[i] = max(v1, v2)
    return new_v

def get_greedy(sp, v):
    """
    Compute optimal actions taking v as the value function.  Parameters are
    the same as for bellman().  Returns a NumPy array called "policy", where
    policy[i] is the optimal action at sp.grid_points[i,:].  The optimal
    action is represented in binary, where 0 indicates reject and 1 indicates
    accept.
    """
    f, g, beta, c, q = sp.f, sp.g, sp.beta, sp.c, sp.q  # Simplify names
    vf = LinearNDInterpolator(sp.grid_points, v)
    N = len(v)
    policy = np.zeros(N, dtype=int)
    for i in range(N):
        w, pi = sp.grid_points[i,:]
        v1 = w / (1 - beta)
        integrand = lambda m: vf(m, q(m, pi)) * (pi * f(m) + (1 - pi) * g(m))
        integral, error = fixed_quad(integrand, 0, sp.w_max)
        v2 = c + beta * integral
        policy[i] = v1 > v2  # Evaluates to 1 or 0
    return policy
```

The module begins by defining a class `searchProblem`, an instance of which stores the parameters and attributes needed to compute optimal actions

The Bellman operator is implemented as `bellman()`, and `get_greedy()` computes an approximate optimal policy from a guess `v` of the value function

We will omit a detailed discussion of the code because *you are about to construct a more efficient solution method*

Before that let's look quickly at solutions computed from this code

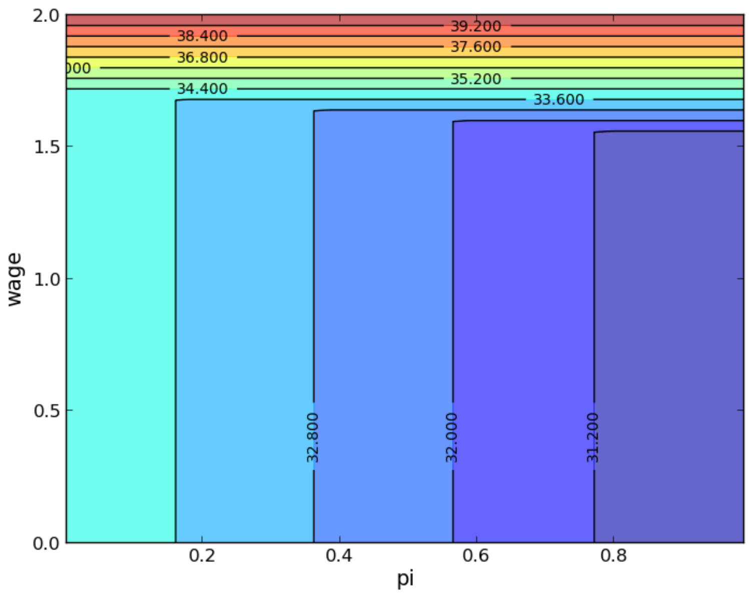
Value function:

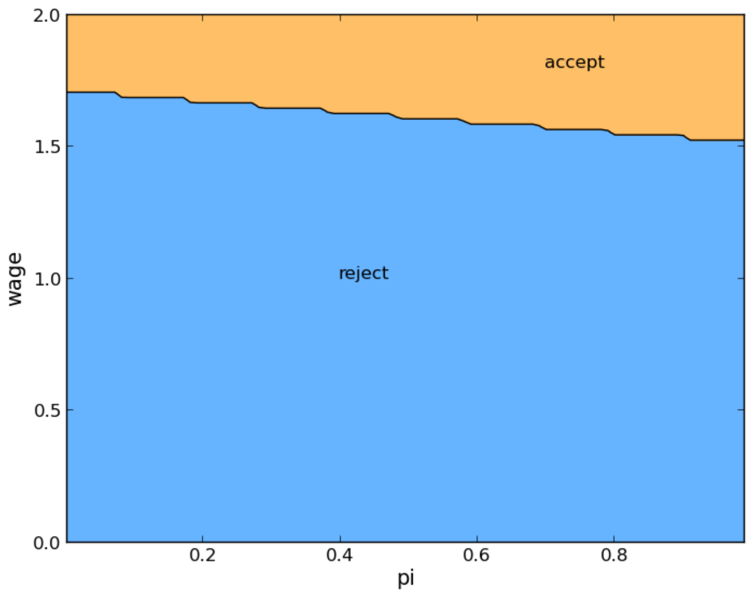
The optimal policy:

Code for producing these figures can be found in file `odu_vfi_plots.py` from the main repository

The results fit well with our intuition from section *Looking Forward*

- black line in the figure above corresponds to the function $\bar{w}(\pi)$ introduced there
- decreasing as expected


39.200
38.400
37.600
36.800
000
35.200
34.400
33.600
32.800
32.000
31.200
wage
pi
2.0
1.5
1.0
0.5
0.0
0.2
0.4
0.6
0.8


accept
reject
wage
pi
2.0
1.5
1.0
0.5
0.0
0.2
0.4
0.6
0.8

Take 2: A More Efficient Method

Our implementation of VFI can be optimized to some degree,

But instead of pursuing that, let's consider another method to solve for the optimal policy

Uses iteration with an operator having the same contraction rate as the Bellman operator, but

- one dimensional rather than two dimensional
- no maximization step

As a consequence, the algorithm is orders of magnitude faster than VFI

Another Functional Equation To begin, note that when $w = \bar w(\pi)$, the worker is indifferent between accepting and rejecting

Hence the two choices on the right-hand side of (1.47) have equal value:

$$\frac{\bar w(\pi)}{1-\beta} = c + \beta \int V(w', \pi') \, h_\pi(w') \, dw' \tag{1.48}$$

Together, (1.47) and (1.48) give

$$V(w, \pi) = \max \left\{ \frac{w}{1-\beta}, \, \frac{\bar w(\pi)}{1-\beta} \right\} \tag{1.49}$$

Combining (1.48) and (1.49), we obtain

$$\frac{\bar w(\pi)}{1-\beta} = c + \beta \int \max \left\{ \frac{w'}{1-\beta}, \, \frac{\bar w(\pi')}{1-\beta} \right\} \, h_\pi(w') \, dw'$$

Multiplying by $1-\beta$, substituting in $\pi' = q(w', \pi)$ and using $\circ$ for composition of functions yields

$$\bar w(\pi) = (1-\beta)c + \beta \int \max \left\{ w', \bar w \circ q(w', \pi) \right\} \, h_\pi(w') \, dw' \tag{1.50}$$

Equation (1.50) can be understood as a functional equation, where $\bar w$ is the unknown function

- Let's call it the *reservation wage functional equation* (RWFE)
- The solution $\bar w$ to the RWFE is the object that we wish to compute

Solving the RWFE To solve the RWFE, we will first show that its solution is the fixed point of a contraction mapping

To this end, let

- $b[0,1]$ be the bounded real-valued functions on $[0,1]$
- $\|\psi\| := \sup_{x \in [0,1]} |\psi(x)|$

Consider the operator Q mapping $\psi \in b[0,1]$ into $Q\psi \in b[0,1]$ via

$$(Q\psi)(\pi) = (1-\beta)c + \beta \int \max \left\{ w', \psi \circ q(w', \pi) \right\} \, h_\pi(w') \, dw' \tag{1.51}$$

Comparing (1.50) and (1.51), we see that the set of fixed points of Q exactly coincides with the set of solutions to the RWFE

- If $Q\bar{w} = \bar{w}$ then $\bar{w}$ that solves (1.50) and vice versa

Moreover, for any $\psi, \phi \in b[0,1]$, basic algebra and the triangle inequality for integrals tells us that

$$|(Q\psi)(\pi) - (Q\phi)(\pi)| \leq \beta \int |\max \{w', \psi \circ q(w', \pi)\} - \max \{w', \phi \circ q(w', \pi)\}| \; h_\pi(w') \, dw' \tag{1.52}$$

Working case by case, it is easy to check that for real numbers a, b, c we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c| \tag{1.53}$$

Combining (1.52) and (1.53) yields

$$|(Q\psi)(\pi) - (Q\phi)(\pi)| \leq \beta \int |\psi \circ q(w', \pi) - \phi \circ q(w', \pi)| \; h_\pi(w') \, dw' \leq \beta \|\psi - \phi\| \tag{1.54}$$

Taking the supremum over π now gives us

$$\|Q\psi - Q\phi\| \leq \beta \|\psi - \phi\| \tag{1.55}$$

In other words, Q is a contraction of modulus β on the complete metric space $(b[0,1], \| \cdot \|)$

Hence

- A unique solution $\bar{w}$ to the RWFE exists in $b[0,1]$
- $Q^k \psi \to \bar{w}$ uniformly as $k \to \infty$, for any $\psi \in b[0,1]$

The following exercise asks you to exploit these facts to compute an approximation to $\bar{w}$

Exercises

Exercise 1 For arbitrary initial condition $\psi \in b[0,1]$, we know that $Q^k \psi \to \bar{w}$ uniformly as $k \to \infty$

Use this fact to compute an approximation to $\bar{w}$ and plot it

Hints:

- Start by implementing Q as a function
- It might be helpful to model this function *loosely* on the function `bellman()` from `odu_vfi.py` — see *above*
- The function `compute_fixed_point()` from the module `compute_fp` is convenient for computing fixed points and can be found in the main repository

Assuming you adopt the default parameters, your result should coincide closely with the figure for the optimal policy *shown above*

Try experimenting with different parameters, and confirm that the change in the optimal policy coincides with your intuition

Solution: *View solution*

References

1.6 Solutions to Exercises

This page collects solutions to all exercises in the course

Note: all of these Python files can be downloaded from the main repository

1.6.1 Exercises from *An Introductory Example*

Solution to *Exercise 1*

```
def factorial(n):
    k = 1
    for i in range(n):
        k = k * (i + 1)
    return k
```

Solution to *Exercise 2*

```
from random import uniform

def binomial_rv(n, p):
    count = 0
    for i in range(n):
        U = uniform(0, 1)
        if U < p:
            count = count + 1     # Or count += 1
    print count
```

Solution to *Exercise 3*

Consider the unit circle embedded in the unit square

Let A be its area and let $r = 1/2$ be its radius

If we know π then we can compute A via $A = \pi r^2$

But here the point is to compute π, which we can do by $\pi = A/r^2$

Summary: If we can estimate the area of the unit circle, then dividing by $r^2 = (1/2)^2 = 1/4$ gives an estimate of π

We estimate the area by sampling bivariate uniforms and looking at the fraction that fall into the unit circle

```
from random import uniform
from math import sqrt

n = 100000

count = 0
for i in range(n):
    u, v = uniform(0, 1), uniform(0, 1)
    d = sqrt((u - 0.5)**2 + (v - 0.5)**2)
    if d < 0.5:
        count += 1
```

```
area_estimate = count / float(n)

print area_estimate * 4  # dividing by radius**2
```

Solution to *Exercise 4*

```
from random import uniform

payoff = 0
count = 0

for i in range(10):
    U = uniform(0, 1)
    count = count + 1 if U < 0.5 else 0
    if count == 3:
        payoff = 1

print payoff
```

Solution to *Exercise 5*

```
from pylab import plot, show
from random import normalvariate

alpha = 0.9
ts_length = 200
current_x = 0

x_values = []
for i in range(ts_length):
    x_values.append(current_x)
    current_x = alpha * current_x + normalvariate(0, 1)
plot(x_values, 'b-')
show()
```

Solution to *Exercise 6*

```
from pylab import plot, show, legend
from random import normalvariate

alphas = [0.0, 0.8, 0.98]
ts_length = 200

for alpha in alphas:
    x_values = []
    current_x = 0
    for i in range(ts_length):
        x_values.append(current_x)
        current_x = alpha * current_x + normalvariate(0, 1)
    plot(x_values, label='alpha = ' + str(alpha))
legend()
show()
```

1.6.2 Exercises from *Python Essentials*

Solution to *Exercise 1*

Part 1 solution

One solution is

```
>>> sum([x * y for x, y in zip(x_vals, y_vals)])
```

Incidentally, this also works

```
>>> sum(x * y for x, y in zip(x_vals, y_vals))
```

Part 2 solution

One solution is

```
>>> sum([x % 2 == 0 for x in range(100)])
```

or just

```
>>> sum(x % 2 == 0 for x in range(100))
```

Some (less natural) alternatives, which help to illustrate the flexibility of list comprehensions, are

```
>>> len([x for x in range(100) if x % 2 == 0])
```

and

```
>>> sum([1 for x in range(100) if x % 2 == 0])
```

Part 3 solution

One solution is

```
>>> pairs = ((2, 5), (4, 2), (9, 8), (12, 10))
>>> sum([x % 2 == 0 and y % 2 == 0 for x, y in pairs])
```

Solution to *Exercise 2*

Solution

```
def p(x, coeff):
    return sum(a * x**i for i, a in enumerate(coeff))
```

Solution to *Exercise 3*

Solution

```python
def f(string):
    count = 0
    for letter in string:
        if letter == letter.upper():
            count += 1
    return count
```

Alternatively,

```python
def f(string):
    return sum(char1 == char2 for char1, char2 in zip(string, string.upper()))
```

Solution to *Exercise 4*

Solution

```python
def f(seq_a, seq_b):
    is_subset = True
    for a in seq_a:
        if a not in seq_b:
            is_subset = False
    return is_subset
```

Of course, if we use the `sets` data type, then the solution is easier

```python
def f(seq_a, seq_b):
    return set(seq_a).issubset(set(seq_b))
```

Solution to *Exercise 5*

```python
def linapprox(f, a, b, n, x):
    """
    Evaluates the piecewise linear interpolant of f at x on the interval [a,
    b], with n evenly spaced grid points.

    Parameters:

        f is a function
        x, a and b are numbers with a <= x <= b
        n is an integer.

    Returns:

        A number (the interpolant evaluated at x)

    """
    length_of_interval = b - a
    num_subintervals = n - 1
    step = length_of_interval / float(num_subintervals)

    # Find the first grid point larger than x
    point = a
    while point <= x:
        point += step
    # Now x must lie between the gridpoints (point - step) and point
    u, v = point - step, point
    return f(u) + (x - u) * (f(v) - f(u)) / (v - u)
```

1.6.3 Exercises from *Object Oriented Programming*

Solution to *Exercise 1*

```
class ecdf:

    def __init__(self, observations):
        self.observations = observations

    def __call__(self, x):
        counter = 0.0
        for obs in self.observations:
            if obs <= x:
                counter += 1
        return counter / len(self.observations)
```

Solution to *Exercise 2*

```
class Polynomial:

    def __init__(self, coefficients):
        """
        Creates an instance of the Polynomial class representing
        p(x) = a_0 x^0 + ... + a_N x^N, where a_i = coefficients[i].
        """
        self.coefficients = coefficients

    def __call__(self, x):
        y = 0
        for i, a in enumerate(self.coefficients):
            y += a * x**i
        return y

    def differentiate(self):
        new_coefficients = []
        for i, a in enumerate(self.coefficients):
            new_coefficients.append(i * a)
        # Remove the first element, which is zero
        del new_coefficients[0]
        # And reset coefficients data to new values
        self.coefficients = new_coefficients
```

Note: if we replace `evaluate` with `__call__` then

```
>>> p.__call__(1) == p(1)
```

This is an example of a *special method*, in this case leading to cleaner code

1.6.4 Exercises from *Advanced Features*

Solution to *Exercise 1*

Here’s the standard solution

```
def x(t):
    if t == 0:
        return 0
    if t == 1:
        return 1
    else:
        return x(t-1) + x(t-2)
```

Solution to *Exercise 2*

One solution is as follows

```
def column_iterator(target_file, column_number):
    """A generator function for CSV files.
    When called with a file name target_file (string) and column number
    column_number (integer), the generator function returns a generator
    which steps through the elements of column column_number in file
    target_file.
    """
    f = open(target_file, 'r')
    for line in f:
        yield line.split(',')[column_number - 1]
    f.close()

dates = column_iterator('test_table.csv', 1)

for date in dates:
    print date
```

Solution to *Exercise 3*

```
f = open('numbers.txt')

total = 0.0
for line in f:
    try:
        total += float(line)
    except ValueError:
        pass

f.close()

print total
```

1.6.5 Exercises from *NumPy*

Solution to *Exercise 1*

```
import numpy as np

def p(coef, x):
    X = np.empty(len(coef))
    X[0] = 1
    X[1:] = x
    y = np.cumprod(X)    # y = [1, x, x**2,...]
    return np.dot(coef, y)
```

Solution to *Exercise 2*

Take your time, read it slowly and you will understand:

```
"""
Origin: QEwP by John Stachurski and Thomas J. Sargent
Date:   2/2013
File:   discreterv.py

"""

from numpy import cumsum
from numpy.random import uniform

class discreteRV:
    """
    Generates an array of draws from a discrete random variable with vector of
    probabilities given by q.  In particular, the draw() method returns i with
    probability q[i].
    """

    def __init__(self, q):
        self.set_q(q)  # q must be array like

    def set_q(self, q):
        self.Q = cumsum(q)   # Cumulative sum

    def draw(self, k=1):
        "Returns k draws from q."
        return self.Q.searchsorted(uniform(0, 1, size=k))
```

Solution to *Exercise 3*

```
"""
Origin: QEwP by John Stachurski and Thomas J. Sargent
Date:   5/2013
File:   ecdf.py

Implements the empirical distribution function.

"""
```

```
import numpy as np
import matplotlib.pyplot as plt

class ecdf:

    def __init__(self, observations):
        self.observations = np.asarray(observations)

    def __call__(self, x):
        return np.mean(self.observations <= x)

    def plot(self, a=None, b=None):
        # Choose a reasonable interval if [a, b] is not specified
        if not a:
            a = self.observations.min() - self.observations.std()
        if not b:
            b = self.observations.max() + self.observations.std()
        x_vals = np.linspace(a, b, num=100)
        f = np.vectorize(self.__call__)
        plt.plot(x_vals, f(x_vals))
        plt.show()
```

1.6.6 Exercises from *SciPy*

Solution to *Exercise 1*

Here's a recursive implementation of the bisection algorithm, in file `bisection2.py`

```
def bisect(f, a, b, tol=10e-5):
    """
    Implements the bisection root finding algorithm, assuming that f is a
    real-valued function on [a, b] satisfying f(a) < 0 < f(b).
    """
    lower, upper = a, b
    if upper - lower < tol:
        return 0.5 * (upper + lower)
    else:
        middle = 0.5 * (upper + lower)
        print('Current mid point = {}'.format(middle))
        if f(middle) > 0:   # Implies root is between lower and middle
            bisect(f, lower, middle)
        else:               # Implies root is between middle and upper
            bisect(f, middle, upper)
```

We can test it as follows

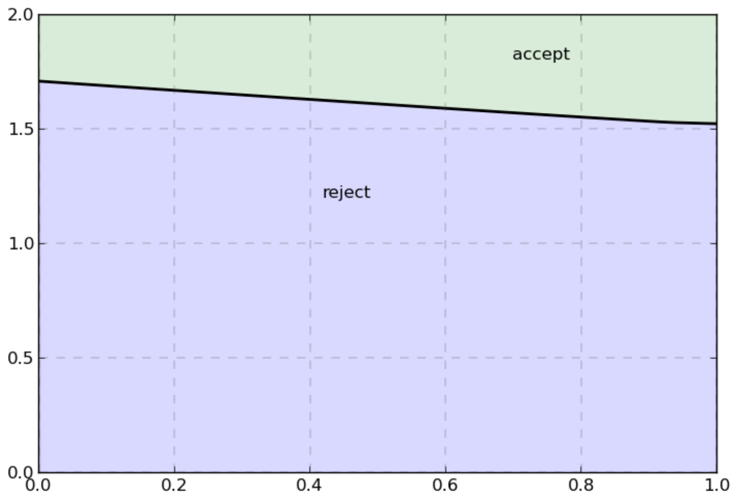
```
In [23]: run bisection2.py

In [24]: f = lambda x: np.sin(4 * (x - 0.25)) + x + x**20 - 1

In [25]: bisect(f, 0, 1)
Current mid point = 0.5
Current mid point = 0.25
Current mid point = 0.375
Current mid point = 0.4375
Current mid point = 0.40625
Current mid point = 0.421875
Current mid point = 0.4140625
Current mid point = 0.41015625
Current mid point = 0.408203125
Current mid point = 0.4091796875
Current mid point = 0.40869140625
Current mid point = 0.408447265625
Current mid point = 0.408325195312
Current mid point = 0.408264160156
```

1.6.7 Exercises from *Search with Offer Distribution Unknown*

Solution to *Exercise 1*

```python
"""
Solves the "Offer Distribution Unknown" model by iterating on a guess of the
reservation wage function.
"""
from scipy import interp
import numpy as np
from numpy import maximum as npmax
import matplotlib.pyplot as plt
from odu_vfi import searchProblem
from scipy.integrate import fixed_quad
from compute_fp import compute_fixed_point


def res_wage_operator(sp, phi):
    """
    Updates the reservation wage function guess phi via the operator Q.
    Returns the updated function Q phi, represented as the array new_phi.

        * sp is an instance of searchProblem, defined in odu_vfi
        * phi is a NumPy array with len(phi) = len(sp.pi_grid)

    """
    beta, c, f, g, q = sp.beta, sp.c, sp.f, sp.g, sp.q    # Simplify names
    phi_f = lambda p: interp(p, sp.pi_grid, phi)  # Turn phi into a function
    new_phi = np.empty(len(phi))
    for i, pi in enumerate(sp.pi_grid):
        def integrand(x):
            "Integral expression on right-hand side of operator"
            return npmax(x, phi_f(q(x, pi))) * (pi * f(x) + (1 - pi) * g(x))
        integral, error = fixed_quad(integrand, 0, sp.w_max)
        new_phi[i] = (1 - beta) * c + beta * integral
    return new_phi


if __name__ == '__main__':  # If module is run rather than imported

    sp = searchProblem(pi_grid_size=50)
    phi_init = np.ones(len(sp.pi_grid))
    w_bar = compute_fixed_point(res_wage_operator, sp, phi_init)

    fig, ax = plt.subplots()
    ax.plot(sp.pi_grid, w_bar, linewidth=2, color='black')
    ax.set_ylim(0, 2)
    ax.grid(axis='x', linewidth=0.25, linestyle='--', color='0.25')
    ax.grid(axis='y', linewidth=0.25, linestyle='--', color='0.25')
    ax.fill_between(sp.pi_grid, 0, w_bar, color='blue', alpha=0.15)
    ax.fill_between(sp.pi_grid, w_bar, 2, color='green', alpha=0.15)
    ax.text(0.42, 1.2, 'reject')
    ax.text(0.7, 1.8, 'accept')
    fig.show()
```

Here's a sample output generated by running this code



You should find that the run time is much shorter than that of the value function approach in `odu_vfi.py`

1.6.8 Exercises from *Modeling Career Choice*

Solution to *Exercise 1*

The sample path figures can be generated with the following code

```
import matplotlib.pyplot as plt
import numpy as np
from discreterv import discreteRV
from career import *
from compute_fp import compute_fixed_point

wp = workerProblem()
v_init = np.ones((wp.N, wp.N))*100
v = compute_fixed_point(bellman, wp, v_init)
optimal_policy = get_greedy(wp, v)
F = discreteRV(wp.F_probs)
G = discreteRV(wp.G_probs)

def gen_path(T=20):
    i = j = 0
    theta_index = []
    epsilon_index = []
    for t in range(T):
        if optimal_policy[i, j] == 1:     # Stay put
            pass
        elif optimal_policy[i, j] == 2:   # New job
            j = int(G.draw())
        else:                             # New life
            i, j  = int(F.draw()), int(G.draw())
        theta_index.append(i)
        epsilon_index.append(j)
    return wp.theta[theta_index], wp.epsilon[epsilon_index]

theta_path, epsilon_path = gen_path()
fig = plt.figure()
ax1 = plt.subplot(211)
ax1.plot(epsilon_path, label='epsilon')
ax1.plot(theta_path, label='theta')
ax1.legend(loc='lower right')
theta_path, epsilon_path = gen_path()
ax2 = plt.subplot(212)
ax2.plot(epsilon_path, label='epsilon')
ax2.plot(theta_path, label='theta')
ax2.legend(loc='lower right')
fig.show()
```

Solution to *Exercise 2*

The median for the original parameterization can be computed as follows

```
import matplotlib.pyplot as plt
import numpy as np
from discreterv import discreteRV
from career import *
from compute_fp import compute_fixed_point

wp = workerProblem()
v_init = np.ones((wp.N, wp.N))*100
v = compute_fixed_point(bellman, wp, v_init)
optimal_policy = get_greedy(wp, v)
F = discreteRV(wp.F_probs)
G = discreteRV(wp.G_probs)

def gen_first_passage_time():
    t = 0
    i = j = 0
    theta_index = []
    epsilon_index = []
    while 1:
        if optimal_policy[i, j] == 1:     # Stay put
            return t
        elif optimal_policy[i, j] == 2:   # New job
            j = int(G.draw())
        else:                             # New life
            i, j  = int(F.draw()), int(G.draw())
        t += 1

M = 25000 # Number of samples
samples = np.empty(M)
for i in range(M):
    samples[i] = gen_first_passage_time()
print np.median(samples)
```

To compute the median with $\beta = 0.99$ instead of the default value $\beta = 0.95$, replace `wp = workerProblem()` with `wp = workerProblem(beta=0.99)`

The medians are subject to randomness, but should be about 7 and 11 respectively. Not surprisingly, more patient workers will wait longer to settle down to their final job

Solution to *Exercise 3*

Here's the code to reproduce the original figure

```
import matplotlib.pyplot as plt
from matplotlib import cm
from career import *
from compute_fp import compute_fixed_point

wp = workerProblem()
v_init = np.ones((wp.N, wp.N))*100
v = compute_fixed_point(bellman, wp, v_init)
optimal_policy = get_greedy(wp, v)

fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111)
tg, eg = np.meshgrid(wp.theta, wp.epsilon)
lvls=(0.5, 1.5, 2.5, 3.5)
ax.contourf(tg, eg, optimal_policy.T, levels=lvls, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, optimal_policy.T, colors='k', levels=lvls, linewidths=2)
ax.set_xlabel('theta', fontsize=14)
ax.set_ylabel('epsilon', fontsize=14)
ax.text(1.8, 2.5, 'new life', fontsize=14)
ax.text(4.5, 2.5, 'new job', fontsize=14, rotation='vertical')
ax.text(4.0, 4.5, 'stay put', fontsize=14)
fig.show()
```

Now we want to set `G_a = G_b = 100` and generate a new figure with these parameters.

To do this we replace

```
wp = workerProblem()
```

with

```
wp = workerProblem(G_a=100, G_b=100)
```

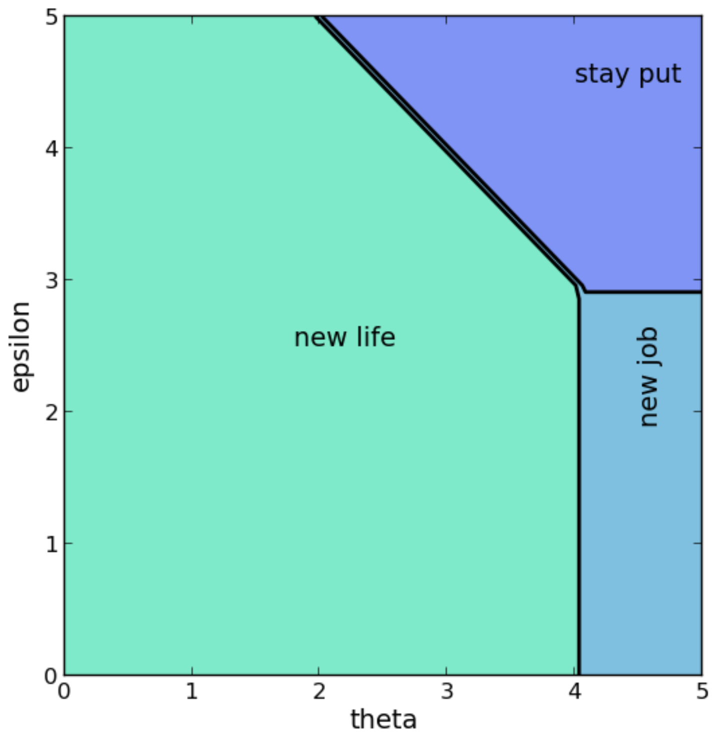
The figure now looks as follows

The region for which the worker will stay put has grown because the distribution for ϵ has become more concentrated around the mean, making high-paying jobs less realistic

1.6.9 Exercises from *On-the-Job Search*

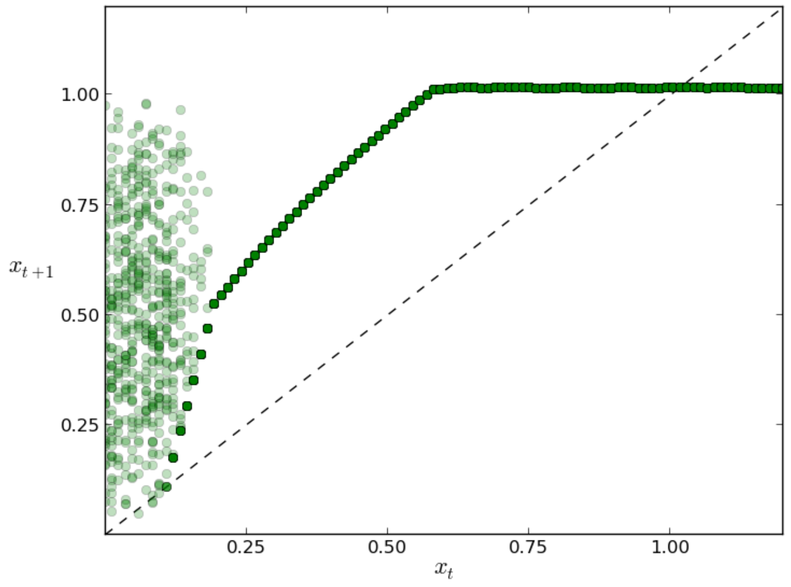
Solution to *Exercise 1*

Here's code to produce the 45 degree diagram



```
import matplotlib.pyplot as plt
import random
from jv import workerProblem, bellman_operator
from compute_fp import compute_fixed_point
import numpy as np

# Set up
wp = workerProblem(grid_size=25)
G, pi, F = wp.G, wp.pi, wp.F       # Simplify names

v_init = wp.x_grid * 0.5
V = compute_fixed_point(bellman_operator, wp, v_init, max_iter=40)
s_policy, phi_policy = bellman_operator(wp, V, return_policies=True)

# Turn the policy function arrays into actual functions
s = lambda y: np.interp(y, wp.x_grid, s_policy)
phi = lambda y: np.interp(y, wp.x_grid, phi_policy)

def h(x, b, U):
    return (1 - b) * G(x, phi(x)) + b * max(G(x, phi(x)), U)

plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
fig, ax = plt.subplots()
ax.set_xlim(0, plot_grid_max)
ax.set_ylim(0, plot_grid_max)
ticks = (0.25, 0.5, 0.75, 1.0)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xlabel(r'$x_t$', fontsize=16)
ax.set_ylabel(r'$x_{t+1}$', fontsize=16, rotation='horizontal')

ax.plot(plot_grid, plot_grid, 'k--')  # 45 degree line
for x in plot_grid:
    for i in range(50):
        b = 1 if random.uniform(0, 1) < s(x) else 0
        U = wp.F.rvs(1)
        y = h(x, b, U)
        ax.plot(x, y, 'go', alpha=0.25)
plt.show()
```

Here's the figure that it produces



Looking at the dynamics, we can see that

- If $x_t$ is below about 0.2 the dynamics are random, but $x_{t+1} > x_t$ is very likely
- As $x_t$ increases the dynamics become deterministic, and $x_t$ converges to a steady state value close to 1

Referring back to the *Optimal policies* figure, we see that $x_t \approx 1$ means that $s_t = s(x_t) \approx 0$ and $\phi_t = \phi(x_t) \approx 0.6$

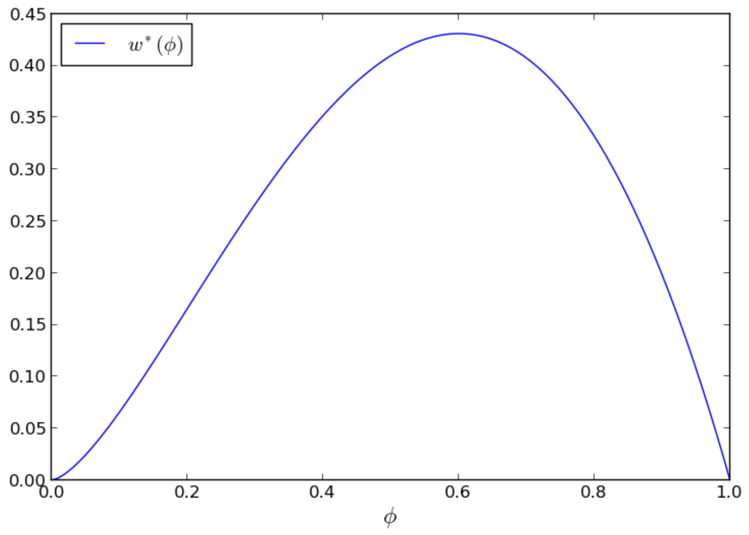
Solution to *Exercise 2*

The figure can be produced as follows

```
from matplotlib import pyplot as plt
from jv import workerProblem
import numpy as np

# Set up
wp = workerProblem(grid_size=25)
```

```
def xbar(phi):
    return (wp.A * phi**wp.alpha)**(1 / (1 - wp.alpha))

phi_grid = np.linspace(0, 1, 100)
fig, ax = plt.subplots()
ax.set_xlabel(r'$\phi$', fontsize=16)
ax.plot(phi_grid, [xbar(phi) * (1 - phi) for phi in phi_grid], 'b-', label=r'$w^*(\phi)$')
ax.legend(loc='upper left')
fig.show()
```

It generates the plot



Observe that the maximizer is around 0.6

This this is similar to the long run value for $\phi$ obtained in exercise 1

Hence the behaviour of the infinitely patent worker is similar to that of the worker with $\beta = 0.96$

This seems reasonable, and helps us confirm that our dynamic programming solutions are probably correct

1.6.10 Exercises from *Optimal Savings*

Solution to *Exercise 1*

```
from matplotlib import pyplot as plt
from ifp import *

m = consumerProblem()
K = 80

# Bellman iteration
V, c = initialize(m)
print "Starting value function iteration"
for i in range(K):
    print "Current iterate = " + str(i)
    V = bellman_operator(m, V)
```

```
c1 = bellman_operator(m, V, return_policy=True)

# Policy iteration
print "Starting policy function iteration"
V, c2 = initialize(m)
for i in range(K):
    print "Current iterate = " + str(i)
    c2 = coleman_operator(m, c2)

fig, ax = plt.subplots()
ax.plot(m.asset_grid, c1[:, 0], label='value function iteration')
ax.plot(m.asset_grid, c2[:, 0], label='policy function iteration')
ax.set_xlabel('asset level')
ax.set_ylabel('consumption (low income)')
ax.legend(loc='upper left')
fig.show()
```

Solution to *Exercise 2*

```
from compute_fp import compute_fixed_point
from matplotlib import pyplot as plt
import numpy as np
from ifp import coleman_operator, consumerProblem, initialize

r_vals = np.linspace(0, 0.04, 4)

fig, ax = plt.subplots()
for r_val in r_vals:
    cp = consumerProblem(r=r_val)
    v_init, c_init = initialize(cp)
    c = compute_fixed_point(coleman_operator, cp, c_init)
    ax.plot(cp.asset_grid, c[:, 0], label=r'$r = %.3f$' % r_val)

ax.set_xlabel('asset level')
ax.set_ylabel('consumption (low income)')
ax.legend(loc='upper left')
fig.show()
```

Solution to *Exercise 3*

```
from matplotlib import pyplot as plt
import numpy as np
from ifp import consumerProblem, coleman_operator, initialize
from compute_fp import compute_fixed_point
from scipy import interp
import mc_sample

def compute_asset_series(cp, T=500000):
    """
    Simulates a time series of length T for assets, given optimal savings
    behavior.  Parameter cp is an instance of consumerProblem
    """

    Pi, z_vals, R = cp.Pi, cp.z_vals, cp.R  # Simplify names
    v_init, c_init = initialize(cp)
    c = compute_fixed_point(coleman_operator, cp, c_init)
    cf = lambda a, i_z: interp(a, cp.asset_grid, c[:, i_z])
    a = np.zeros(T+1)
    z_seq = mc_sample.sample_path(Pi, sample_size=T)
    for t in range(T):
        i_z = z_seq[t]
        a[t+1] = R * a[t] + z_vals[i_z] - cf(a[t], i_z)
    return a

if __name__ == '__main__':

    cp = consumerProblem(r=0.03, grid_max=4)
    a = compute_asset_series(cp)
    fig, ax = plt.subplots()
    ax.hist(a, bins=20, alpha=0.5, normed=True)
    ax.set_xlabel('assets')
    ax.set_xlim(-0.05, 0.75)
    fig.show()
```

Solution to *Exercise 4*

```
from matplotlib import pyplot as plt
import numpy as np
from compute_fp import compute_fixed_point
from ifp import coleman_operator, consumerProblem, initialize
from solution_ifp_ex3 import compute_asset_series

M = 25
r_vals = np.linspace(0, 0.04, M)
fig, ax = plt.subplots()

for b in (1, 3):
    asset_mean = []
    for r_val in r_vals:
        cp = consumerProblem(r=r_val, b=b)
        mean = np.mean(compute_asset_series(cp, T=250000))
        asset_mean.append(mean)
    ax.plot(asset_mean, r_vals, label=r'$b = %d$' % b)

ax.set_yticks(np.arange(.0, 0.045, .01))
ax.set_xticks(np.arange(-3, 2, 1))
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.grid(True)
ax.legend(loc='upper left')
fig.show()
```

1.7 Resources

This page collects some useful links and commands

1.7.1 Useful Links

- http://www.wakari.io — cloud computing with IPython Notebook interface
- https://github.com/jstac/quant-econ — our public code repository

1.7.2 Starting Python

Commands for the system terminal (command prompt)

- `python` — basic Python shell
- `spyder` — integrated development environment for Python
- `ipython` — a better Python shell
- `ipython notebook --pylab inline` — start IPython Notebook on local machine

1.7.3 IPython Notebook

Common commands

- `pwd` — show present working directory
- `ls` — list contents of present working directory
- `cd dir_name` — change to directory `dir_name`
- `cd ..` — go back
- `loadpy file_name.py` — load `file_name.py` into cell
- `%%file new_file.py` — put at top of cell to save contents as `new_file.py`

1.7.4 Git

- To install Git, visit https://github.com/
- To download the public repository run `git clone https://github.com/jstac/quant-econ` in the system terminal
  - or `!git clone https://github.com/jstac/quant-econ` within IPython

1.7.5 PDF Lectures

- `Lecture 1`
- `Lecture 2`
- `Lecture 3`

BIBLIOGRAPHY

[Neal1999] Neal, D. (1999). The Complexity of Job Mobility among Young Men, *Journal of Labor Economics,* 17(2), 237-261.

[Aiyagari1994] Aiyagari, S. R. (1994). Uninsured idiosyncratic risk and aggregate saving, *Quarterly Journal of Economics,* 109(3), 569-584.

[Coleman1990] Coleman, J. W., (1990). Solving the stochastic growth model by policy-function iteration, *Journal of Business and Economic Statistics,* 8(1), 27–29.

[Deaton1991] Deaton, A. (1991). Saving and liquidity constraints, *Econometrica*, 59(5), 1221–1248.

[DenHaan2010] Den Haan, W. J. (2010). Comparison of solutions to the incomplete markets model with aggregate uncertainty, *Journal of Economic Dynamics and Control,* 34(1), 4–27.

[HopenhaynPrescott1992] Hopenhayn, Hugo A. and Edward C. Prescott (1992). Stochastic monotonicity and stationary distributions for dynamic economies, *Econometrica*, 60, 1387–1406.

[Huggett1993] Huggett, M. (1993). The risk-free rate in heterogeneous-agent incomplete-insurance economies, *Journal of economic Dynamics and Control*, 17(5), 953–969.

[Kuhn2013] Kuhn, M. (2013). Recursive equilibria in an Aiyagari-style economy with permanent income shocks, *International Economic Review,* in press.

[Rabault2002] Rabault, G. (2002). When do borrowing constraints bind? Some new results on the income fluctuation problem, *Journal of Economic Dynamics and Control,* 26(2), 217–245.

[Reiter2008] Reiter, M. (2008). Solving heterogeneous-agent models by projection and perturbation, Working paper.

[SchechtmanEscudero1977] Schechtman, J. and V. L. S. Escudero (1977). Some results on an income fluctuation problem, *Journal of Economic Theory,* 16, 151–166.

[CryerChan2008] Cryer, J. D. and K-S. Chan (2008). *Time Series Analysis,* 2nd edition, Springer

[Sargent1987] Sargent, Thomas J. (1987). *Macroeconomic Theory,* 2nd edition, Academic Press

[Shiryaev1995] Shiryaev, A. A. (1995). *Probability*, 2nd edition, Springer

[McCall1970] McCall, J. J. (1970). Economics of Information and Job Search, *Quarterly Journal of Economics,* 84(1), 113-126.