The following is an excerpt from book I am writing for 6.00. The book (alas) is well behind schedule. I can't promise that there are no errors in this material. However, I thought that it would be helpful to you.

The first section covers some aspects of using Pylab to generate plots. The second section contains an extended example intended to illustrate some aspects of using classes that you haven't seen. It also contains more examples of ways to construct plots. Don't worry about understanding the details about mortgages and discounted cash flow (unless you are about to buy a house).

I would be delighted to receive suggestions about how to improve the presentation of this material, especially any errors that you might find.

-- John Guttag

PL.2. Plotting Using Pylab

Pylab is a Python library that provides many of the facilities of MATLAB, "a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation."[1] Later in the book, we will look at some of the more advanced features of pylab, but in this chapter we focus on some of its facilities for plotting data. The web site http://matplotlib.sourceforge.net/users/index.html contains a complete user's guide for pylab. There are also a number of web sites with excellent tutorials. We will not try to provide a user's guide or a complete tutorial here. Instead, we will merely provide a few example plots and explain the code that generated them. For more details about plotting see http://matplotlib.sourceforge.net.

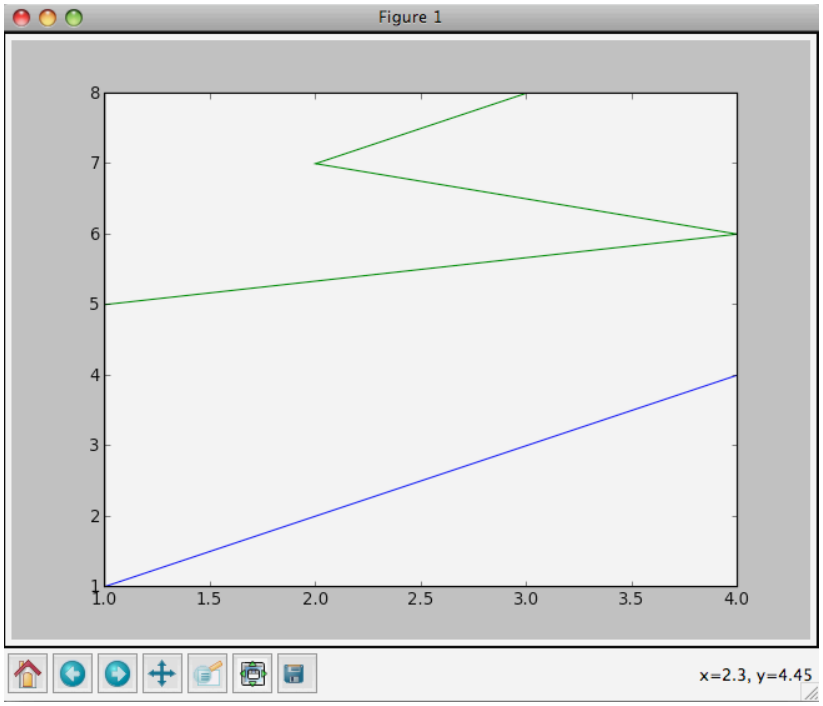
Let's start with something very simple. Executing the code

```
import pylab

pylab.plot([1,2,3,4], [1,2,3,4])
pylab.plot([1,4,2,3], [5,6,7,8])
pylab.show()
```

will cause a window to appear on your computer monitor. Its exact appearance will depend on the operating system on your machine, but it will always look something like:

[1] http://www.mathworks.com/products/matlab/description1.html?s_cid=ML_b1008_desintro



The bar at the top contains the name of the window, in this case "Figure 1." In this example, the name is generated automatically by pylab.

The middle section of the window contains the plot generated by the two lines of code following the import statement. The bottom line was generated by the statement `pylab.plot([1,2,3,4], [1,2,3,4])`. Notice that the two arguments are lists of the same length. Together, they provide a sequence of four <x, y> coordinate pairs, [(1,1), (2,2), (3,3), 4,4)]. These are plotted in order, and then connected by a line. (Pylab automatically chooses the line color for each plot command, but this can be overridden by an optional argument, as we shall see.) The zig zag plot at the top of the middle section is produced the same way. It zig zags because the sequence of Cartesian points being connected is [(1,5), (4,6), (2,7), (3,8)].

The final line of code, pylab.show() causes the window to appear on the screen. If that line were not there, the figure would still have been produced, but it would not have been displayed. This is not as silly as it at first sounds, since one might well choose to write the figure to a file rather than display it.[2]

The bar at the bottom of the window contains a number of push buttons. The rightmost button is used to write the plot to a file.[3] The next button over is used to adjust the appearance of the plot in the window. This is useful when there is more than one plot per figure. The next four buttons

[2] In some operating systems, `pylab.show()` causes the process running Python to be suspended. This is unfortunate. The usual work around is to put the command as the last line to be executed.

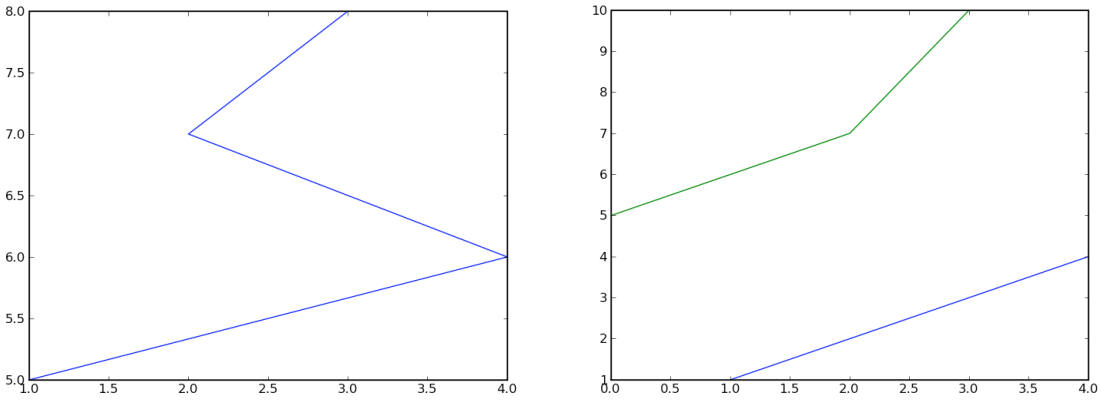
[3] For those of you too young to know, the icon represents a "floppy disk." Floppy disks were first introduced by IBM in 1971. They were eight inches in diameter and held all of 80K bytes. Unlike later floppy disks, they were actually floppy. The original IBM PC had a single 160KB 5.5 inch floppy disk drive. For most of the 1970's and 1980's, floppy disks were the primary storage device for personal computers. The transition to rigid cases (as represented in the icon that launched this digression) started in the mid-1980's (with the Macintosh), which didn't stop people from calling them floppy disks.

are used for panning and zooming. And the button on the left is used to restore the figure to its original appearance after you are done playing with pan and zoom.

Of course, it is possible to produce more than one figure and to write them to files rather than the display. The following code

```
pylab.figure(1)
pylab.plot([1,2,3,4], [1,2,3,4])
pylab.figure(2)
pylab.plot([1,4,2,3], [5,6,7,8])
pylab.savefig('firstSaved')
pylab.figure(1)
pylab.plot([5,6,7,10])
pylab.savefig('secondSaved')
```

produces and saves to files named `firstSaved.png` and `secondSaved.png` the two plots:



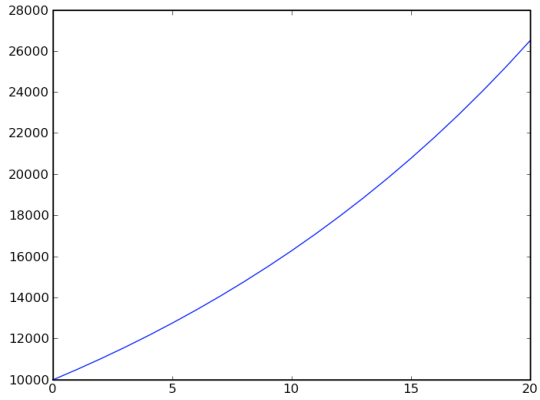
Pylab has a notion of "current figure." Executing the statement `pylab.figure(x)` sets the current figure to the figure numbered `x`. Subsequently executed pylab commands implicitly refer to that figure, until another `pylab.figure` command is executed. This explains why the plot on the left, which corresponds to the figured numbered 2, is stored in `firstSaved.png`.

Observe that the last call to `pylab.plot` is passed only one argument. This argument supplies the Y values. The corresponding X values are `range(len(Y-values))`, which is why they range from 0 to 3 in this case.

Let's look at another example. The code

```
principal = 10000 #initial investment
interestRate = 0.05
years = 20
values = []
for i in range(years + 1):
    values.append(principal)
    principal += principal*interestRate
pylab.plot(values)
```

produces the plot

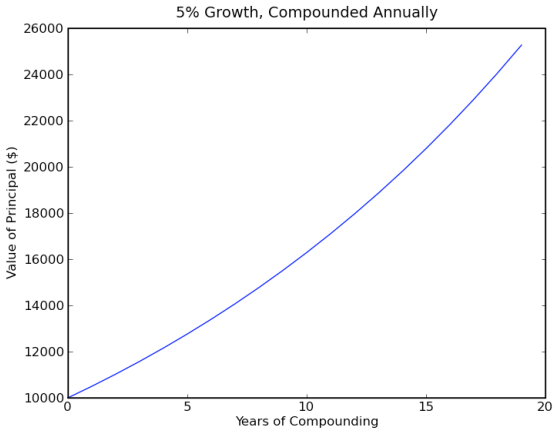


If we look at the code, we can deduce that this is a plot showing the growth of an initial investment of $10,000 with at an annually compounded interest rate of 5%. However, this cannot be easily inferred looking only at the plot itself. That's a bad thing. All plots should have informative titles, and all axes should be labeled.

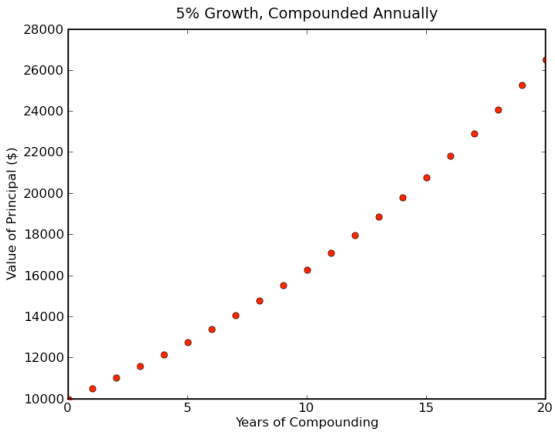
If we add to the end of our the code the following lines

```
pylab.title('5% Growth, Compounded Annually')
pylab.xlabel('Years of Compounding')
pylab.ylabel('Value of Principal ($)')
```

we get the more informative plot



For every plotted curve, there is an optional argument that is a format string indicating the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and are composed of a color indicator followed by line style indicator. The default format string is 'b-', which, as we have seen is a solid blue line. To plot the above with red circles, you would replace the call to `pylab.plot` by `pylab.plot(values, 'ro')`, which produces the plot:



See http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot for a complete list of line styles and format strings

PL.2 Mortgages

In the fall of 2008 in the United States, a collapse in housing prices helped trigger a severe economic melt down. One of the contributing factors was that many home owners had taken on mortgages that ended up having unexpected consequences.[4]

In the beginning, mortgages were relatively simple beasts. One borrowed money from a bank and make a fixed size payment each month for the life of the mortgage, which typically ranged from fifteen to thirty years. At the end of that period, the bank had been paid back the initial loan plus interest.

Towards the end of the twentieth century, mortgages started getting a lot more complicated. People could get lower interest rates by paying "points" at the time they took on the mortgage. A point is a cash payment of 1% of the value of the loan. People could take mortgages that were "interest only" for a period of time. That is to say for some number of months at the start of the loan the borrower paid only the accrued interest and none of principal. Other loans involved multiple rates. Typically the initial rate (called a "teaser rate") was low, and then it went up over time. Many of these loans were variable rate, i.e., the rate to be paid after the initial period would vary depending upon interest rates.

In principal, giving consumers a variety of options is a good thing. However, unscrupulous loan purveyors were not always careful to fully explain the possible long-term implications of the various options, and some naïve borrowers made choices that proved to have dire consequences. There was no need for this, since it is relatively easy to write a program (and many such programs exist) that plots the implications of various kinds of loans.

Let's build a program that plots the costs of four kinds of loans:

- A fixed rate mortgage with no points,

[4] In this context, it is worth recalling that etymology of the word mortgage. *The American Heritage Dictionary of the English Language* traces the word back to the old French words for dead (*mort*) and pledge (*gage*). (This derivation also explains why the "t" in the middle of mortgage is silent.)

- A fixed rate mortgage with points,
- A fixed rate mortgage with an initial interest only period, and
- A mortgage with an initial teaser rate followed by a higher rate for the duration.

The points of this exercise are to 1) provide some experience in the incremental development of a set of related classes, and 2) get some practice in the visual presentation of data.

We will structure our code to include a `Mortgage` base class, and subclasses corresponding to each of the four kinds of mortgages listed above. Figure MortgageBase contains an **abstract class** `Mortgage`. This class contains methods that are shared by each of the subclasses, but it not intended to be instantiated directly.

```
def findPayment(loan, r, m):
    return loan*((r*(1+r)**m)/((1+r)**m - 1))

class Mortgage(object):
    def __init__(self, loan, r, months):
        self.loan = loan
        self.rate = r/12.0
        self.months = months
        self.payments = [0.0]
        self.owed = [loan]
        self.payment = findPayment(loan, self.rate, months)
        self.legend = None
    def getTot(self):
        return sum(self.payments)
    def getLegend(self):
        return self.legend
    def makePayment(self):
        self.payments.append(self.payment)
        reduction = self.payment - self.owed[-1]*self.rate
        self.owed.append(self.owed[-1] - reduction)
```

Figure MortgageBase

Looking at `__init__`, we see that all `Mortgage` instances will have attributes corresponding to the initial loan amount, the monthly interest rate, the duration of the loan in months, a list of payments that have been at the start of each month (the list starts with `0.0`, since no payments have been made at the start of the first month), the balance of the initial loan that is outstanding at the start of each month, the amount of money to paid each month, and a legend (which initially has a value of `None`). The `__init__` operation of each subclass is expected to start by calling `Mortgage.__init__`, and then to initialize `self.legend` to an appropriate description of that subclass.

The field `self.payment` is initialized using the function `findPayment`, found at the top of the figure. Some subclasses of `Mortgage` may reinitialize this to a different initial value. The function `findPayment` computes the size of the fixed monthly payment needed to pay off the loan, including interest, by the end of its term. It does this using a well know closed form expression. This expression is not hard to derive, but it is a lot easier to just look it up and more

likely to be correct than one derived on the spot. When you write code that incorporates formulas you have looked up, make sure that

1. You have taken the formula from a reputable source. We looked at multiple reputable sources, all of which contained equivalent formulas,
2. You fully understand the meaning of the all the variables in the formula, and
3. You test your implementation against examples taken from reputable sources. After implementing this, we tested it by comparing our results to the results supplied by a mortgage calculator available on the web.

The method `makePayment` is used to record mortgage payments. Part of each payment covers the amount of interest due on the outstanding loan balance, the remainder of the payment is used to reduce the loan balance. That is why `makePayment` updates both `self.payments` and `self.owed`.

Figure FixedRate contains classes implementing two of the mortgage types. Each of these classes overrides `__init__` so that it can set `legend` to an appropriate value, and inherits the other three methods from `Mortgage`.

```
class Fixed(Mortgage):
    def __init__(self, loan, r, months):
        Mortgage.__init__(self, loan, r, months)
        self.legend = 'Fixed, ' + str(r*100) + '%'

ass FixedWithPts(Mortgage):
    def __init__(self, loan, r, months, pts):
        Mortgage.__init__(self, loan, r, months)
        self.pts = pts
        self.payments = [loan*(pts/100.0)]
        self.legend = 'Fixed, ' + str(r*100) + '%, '\
                      + str(pts) + ' points'
```

Figure FixedRate

The class `IntOnly`, Figure IntOnly, treats this class of mortgages as equivalent to the concatenation of two separate loans. Since during the first `teaserMonths` payments do not reduce the outstanding principal, `__init__` initialized `self.payment` to exactly the amount needed to pay the interest. After teaserMonths payments have been made, `makePayment` resets `self.payment` to an amount sufficient to payoff the loan over the remaining months.

```
class IntOnly(Mortgage):
    def __init__(self, loan, r, months, teaserMonths):
        Mortgage.__init__(self, loan, r, months)
        self.teaserMonths = teaserMonths
        self.payment = loan*r/12.0
        self.legend = 'Fixed, ' + str(r*100) + '%, '\
                      + 'Interest only for '\
                      + str(teaserMonths) + ' months'
    def makePayment(self):
        if len(self.payments) == self.teaserMonths + 1:
            self.payment = findPayment(self.loan, self.rate,
                                       self.months - self.teaserMonths)
        Mortgage.makePayment(self)
```

Figure IntOnly

The class `TwoRate`, Figure TwoRate, also treats the mortgage as the concatenation of two loans. In this case, however, both loans reduce the outstanding principal.

```
class TwoRate(Mortgage):
    def __init__(self, loan, r, months, teaserRate, teaserMonths):
        Mortgage.__init__(self, loan, teaserRate, months)
        self.teaserMonths = teaserMonths
        self.teaserRate = teaserRate
        self.nextRate = r/12.0
        self.legend = 'Variable, ' + str(teaserRate*100)\
                      + '% for ' + str(self.teaserMonths)\
                      + ' months, then ' + str(r*100) + '%'
    def makePayment(self):
        if len(self.payments) == self.teaserMonths + 1:
            self.rate = self.nextRate
            self.payment = findPayment(self.owed[-1], self.rate,
                                       self.months - self.teaserMonths)
        Mortgage.makePayment(self)
```

Figure TwoRate

Figure TestMortgages contains a function that computes and prints the total cost of each kind of mortgage for a sample set of parameters. It begins by creating one mortgage of each kind. It then makes a monthly payment on each for (assuming the default parameter values) thirty years. Finally, it prints the total amount of the payments for each loan, as seen in the bottom of the figure.

```
def test(amt=200000, years=30, fixedRate=0.07, pts = 3.25,
         ptsRate=0.0625, intOnlyPeriod=60, varRate1=0.04,
         varRate2=0.11, varMonths=48):
    totMonths = years*12
    fixed1 = Fixed(amt, fixedRate, totMonths)
    fixed2 = FixedWithPts(amt, ptsRate, totMonths, pts)
    intOnly = IntOnly(amt, fixedRate, totMonths, intOnlyPeriod)
    twoRate = TwoRate(amt, varRate2, totMonths, varRate1, varMonths)
    morts = [fixed1, fixed2, intOnly, twoRate]
    for m in range(totMonths):
        for mort in morts:
            mort.makePayment()
    for m in morts:
        print m.getLegend()
        print '   Total payments = $' + str(int(m.getTot()))
```

```
>>> test()
Fixed, 7.0%
   Total payments = $479017
Fixed, 6.25%, 3.25 points
   Total payments = $443316
Fixed, 7.0%, Interest only for 60 months
   Total payments = $494067
Variable, 4.0% for 48 months, then 11.0%
   Total payments = $607603
```

Figure TestMortgages

At first blush, the results look pretty conclusive. The variable rate loan is a bad idea (for the borrower, not the bank) and the fixed rate loan with points costs the least. It's important to note, however, that total cost is not the only metric by which mortgages should be judged. For example, a borrower who expects his or her income to increase may be willing to pay more in the out years to lesson the burden of payments in the beginning.

This suggests that rather than looking at a single number, we should look at payments over time. This in turn suggests that our program should be producing plots designed to show how the mortgage behaves over time. The class MortgagePlots, Figure MortgagePlots, contains methods that produce such plots.

This class is an example of what is sometimes called a "**mix-in class**."[5] Up to now, we have talked about classes as types and sub-classing as mechanism for defining subtypes. So, for example, we might say that Fixed is a subtype of Mortgage. This is indeed the primary use of classes in well organized Python programs. Occasionally, however, it is convenient to use the class mechanism to bundle together a set of related functions that don't constitute a type. The **multiple inheritance** mechanism of Python can then be used to add the methods defined in the mix-in class to one or more other classes. Here we add it to the Mortgage class, as seen at the bottom of the figure. We think of Mortgage as a subtype of object, but not as a subtype of MortgagePlots. However, this cannot be deduced from the code.

[5] This use of the phrase mix-in can be traced to the Flavors system on the Symbolics Lisp Machine, and (allegedly) to the proximity of Symbolics to an ice cream store famous (at least locally) for its mix-ins.

```
def dcf(val, d, time):
    return val/(1 + d)**time

class MortgagePlots(object):
    def plotPayments(self, color):
        pylab.plot(self.payments, color, label = self.legend)
    def plotTotPd(self, color):
        totPd = [self.payments[0]]
        for i in range(1, len(self.payments)):
            totPd.append(totPd[-1] + self.payments[i])
        pylab.plot(totPd, color, label = self.legend)
    def plotDCF(self, rate, color):
        dcfs = [self.payments[0]]
        for i in range(1, len(self.payments)):
            dcfs.append(dcf(self.payments[i], rate, i))
        cumDCF = []
        for i in range(len(dcfs)):
            cumDCF.append(sum(dcfs[:i]))
        pylab.plot(cumDCF, color, label = self.legend)
    def plotOwed(self, color):
        pylab.plot(self.owed, color, label = self.legend)

class Mortgage(MortgagePlots, object):
   ***Rest of class is unchanged***
```

Figure MortgagePlots

Generally speaking, multiple inheritance is a dangerous mechanism and should only be used in very constrained ways. There are some rules of thumb that can help one stay out of trouble when using multiple inheritance. Consider the case where class `A` has the super classes `B` and `C`.

- It should feel comfortable to say that `A` is a subtype of at most one of `B` and `C`. By convention, it is convenient to list this class last, as we have done in `Mortgage`.
- It should not matter if one writes `class A(B, C)` or class `A(C, B)`, even though Python does specify that these have a different semantics. Roughly speaking, this guideline boils down to saying that the if `a` is an instance of `A`, the text `a.name` should resolve to the same attribute independently of the order in which the interpreter searches the names spaces associated with `B` and `C`.

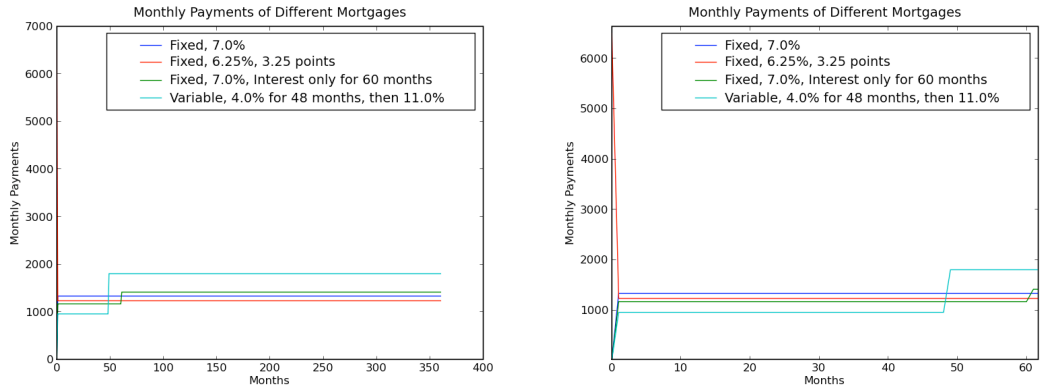
The methods in `MortgagePlot` use a form of `pylab.plot` that we have not yet seen. Each associates a **label** with the generated plot. This is a `str` that can be used as part of a **legend** that can be added to a figure. This, and other, keyword arguments must follow any format strings.

Figure GenPlots contains a function that can be called at the end of the function test (see Figure TestMortgages) to generate four plots intended to provide insight in the different kinds of mortgages. It generates appropriate titles and axis labels for each plot, and then uses the methods in `MortgagePlots` to produce the actual plots. It uses calls to `pylab.figure` to ensure that the appropriate plots appear in the same figure and the index, `i`, into the list `morts` to ensure that he use of colors is consistent across figures.

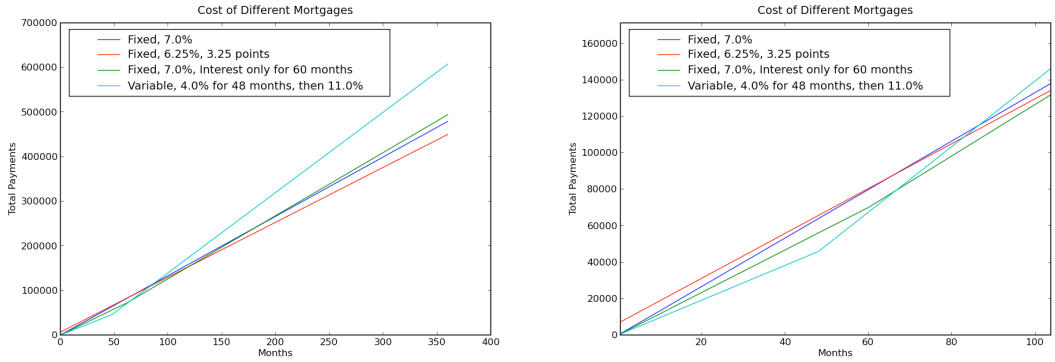
```
def plotMorts(morts, discountRate):
    colors = ['b', 'r', 'g', 'c', 'm', 'k']
    payments = 0
    cost = 1
    dcfPayments = 2
    owed = 3
    pylab.figure(payments)
    pylab.title('Monthly Payments of Different Mortgages')
    pylab.xlabel('Months')
    pylab.ylabel('Monthly Payments')
    pylab.figure(cost)
    pylab.title('Cost of Different Mortgages')
    pylab.xlabel('Months')
    pylab.ylabel('Total Payments')
    pylab.figure(dcfPayments)
    pylab.title('DCF of Different Mortgages (Discount rate='
                + str(discountRate*12*100) + '%)')
    pylab.xlabel('Months')
    pylab.ylabel('Total Discounted Value of Payments')
    pylab.figure(owed)
    pylab.title('Amount Owed on Mortgage')
    pylab.xlabel('Months')
    pylab.ylabel('Amount Owed Bank')
    for i in range(len(morts)):
        pylab.figure(payments)
        morts[i].plotPayments(colors[i])
        pylab.figure(cost)
        morts[i].plotTotPd(colors[i])
        pylab.figure(dcfPayments)
        morts[i].plotDCF(discountRate, colors[i])
        pylab.figure(owed)
        morts[i].plotOwed(colors[i])
    pylab.figure(payments)
    pylab.legend(loc = 'best')
    pylab.figure(cost)
    pylab.legend(loc = 'upper left')
    pylab.figure(dcfPayments)
    pylab.legend(loc = 'lower right')
    pylab.figure(owed)
    pylab.legend(loc = 'lower left')
    pylab.show()
```

Figure MortgagePlots

The method `plotPayments` simply plots each payment against time. The entire plot is on the left, and of the left edge of the plot is on the right. These plots make it clear how the monthly payments vary (or don't) over time, but don't shed much light on the relative costs of each kind of mortgage.



The method `plotTotPd` sheds more light on the cost of each kind of mortgage by plotting the cumulative costs that have been incurred at the start of each month. The entire plot is on the left, and a blowup of the left part of the plot on the right. Looking at where the curves cross suggests that if one plans on holding the mortgage for more than ten years, the fixed mortgage with points will cost the least. On the other hand, if one plans on paying of the mortgage in fewer than sixty-five months, the variable rate mortgage looks like the most cost-effective. Between these two points, the loan that is initially interest only appears to be the best.

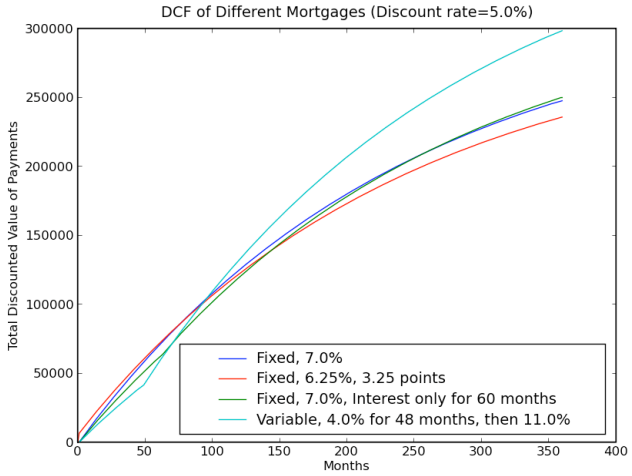


The most interesting method in the class is `plotNPV`. This method plots the **discounted cash flow (DCF)** of the stream of mortgage payments. Any rational person knows $1,000 in hand today is far better than the guarantee of $1,000 thirty years from now. DCF is one way to quantify how much better. DCF is typically used to estimate the attractiveness of an investment opportunity. For example, if the bank is contemplating loaning someone $200,000 it needs to decide if the present value of the expected stream of future payments on that loan exceeds $200,000. The value of such a stream can be estimated by the formula:

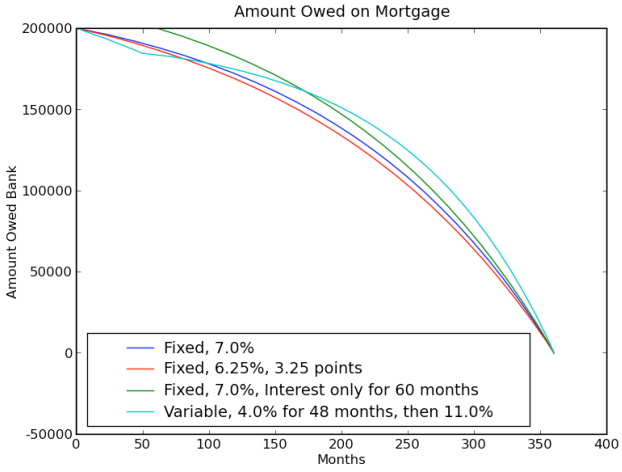
$$DCF = \frac{P_0}{(1+d)^0} + \frac{P_1}{(1+d)^1} + ... + \frac{P_n}{(1+d)^n}$$

where P_x is the payment at time x and d is the discount rate, i.e., an estimate of the instantaneous value of money, e.g., the return that one might expect to get if one invested the same amount of money in an almost risk free instrument (e.g., a government bond).[6]

The plot produced by `plotDCF` is quite different from that produced by `plotTotPd`. Notice how the curves flatten out towards the right side, and how much lower the discounted costs are than the absolute costs. In fact, given the rather high discount rate used, on a discounted cash flow basis the cost of borrowing $200,000 for thirty years can be held to around $25,000. (The DCF of the fixed rate with points minus the $200,000 initially borrowed.) Note that if one uses a lower discount rate, say 1%, the curves look very different.



The last figure plots the outstanding balance left on each kind of loan. As expected each starts at $200,000 and ends at $0, but there are non-negligible differences in the middle.



[6] While DCF is widely used in practice, it is far from perfect. This formulation does not incorporate any risk in the payment stream, e.g., the risk that the borrower may be late with payments or default. Also, it is highly sensitive to the parameter `d`. Small changes in `d` will generate large differences in the DCF, and the risk free rate of return and the inflation rate are both notoriously hard to forecast. Beware of garbage in, garbage out (GIGO).