Unit 7

Learn something new

1. The Consumer Price Index (CPI) we looked at in the last unit is quite volatile from year to year. Let's take a longer-term perspective, looking at changes over five- and ten-year periods by producing a table that looks like this:

```
     Percent increase in CPI
         (Jan-Jan)
    1 year  5 years 10 years
          .
          .
          .
1940    -0.7     2.2    -18.7
1941     1.4     2.2    -11.3
          .
          .
          .
```

   The table shows, for example, that during the ten-year period ending in January 1940 (i.e. from January 1930 to January 1940), the CPI fell 18.7 percent.

   Computing the information in this table is easy enough. Here's code to do the job. Note that we don't always have enough data to go back 5 or 10 years—we use an `if` statement to check. Also, we make use of the `pctIncrease` function written for Program 6.4.

```
import shelve
shelf = shelve.open('cpi')
cpi = shelf['cpi']
shelf.close()

def pctIncrease(begin, end):
   return 100*(end/begin-1)

print('      Percent increase in CPI')
print('           (Jan-Jan)')
print('     1 year  5 years 10 years')

for year in range(1914, 2009):
   currentCPI = cpi[year][1]
   print(year, pctIncrease(cpi[year-1][1], currentCPI))
   if year-5 >= 1913:
      print(pctIncrease(cpi[year-5][1], currentCPI))
   if year-10 >= 1913:
      print(pctIncrease(cpi[year-10][1], currentCPI))
```

The output looks like this. All the information we want is here, but it's a mess. To get a neat table we need to exert much more control over the form in which results are printed.

```
    Percent increase in CPI
        (Jan-Jan)
    1 year  5 years 10 years
          .
          .
          .
1940 -0.714285714286
2.20588235294
-18.7134502924
1941 1.43884892086
2.17391304348
-11.320754717
          .
          .
          .
```

We've been using the print function in its simplest form. The items we pass in—the **arguments**, in technical language—are displayed in the output separated by a single space. After the last item a newline character is added to move the cursor to the beginning of the next line. The two function calls

```
print('Base', 'ball')
print('is an American sport')
```

produce this output

```
Base ball
is an American sport
```

If we like, we can specify our own string to use as a separator as well as one to be used at the end, after the last item, by passing special **keyword arguments** to print. The pair of calls

```
print('Base', 'ball', sep='', end=' ')
print('is an American sport', end='.\n')
```

produces the output

```
Baseball is an American sport.
```

The first **print** uses the empty string as a separator so that 'Base' and 'ball' are conjoined. It ends with a space rather than a newline character. The second **print** ends with both a period and an explicit newline.

Using the keyword arguments **sep** and **end**, we can get all the items for our CPI table on the correct lines. Here's the revised code. The last call to **print** moves us to the next line after we've completed a row of the table. In this final call, we're using the **default arguments** for **print**—**sep=' '** and **end='\n'**—the ones we get if we don't specify otherwise. Without the last call to **print**, everything would be displayed on one very long line, since we've eliminated the default newlines in the other calls to **print**.

```
import shelve
shelf = shelve.open('cpi')
cpi = shelf['cpi']
shelf.close()

def pctIncrease(begin, end):
   return 100*(end/begin-1)

print('      Percent increase in CPI')
print('           (Jan-Jan)')
print('     1 year  5 years 10 years')

for year in range(1914, 2009):
   currentCPI = cpi[year][1]
   print(year, pctIncrease(cpi[year-1][1], currentCPI),
       sep='    ', end='  ')
   if year-5 >= 1913:
      print(pctIncrease(cpi[year-5][1], currentCPI), end='  ')
   if year-10 >= 1913:
      print(pctIncrease(cpi[year-10][1], currentCPI), end='')
   print()
```

Now the output looks like this:

```
     Percent increase in CPI
          (Jan-Jan)
    1 year  5 years 10 years
1914   2.04081632653
1915   1.0
          .
          .
          .
1940   -0.714285714286  2.20588235294  -18.7134502924
1941   1.43884892086  2.17391304348  -11.320754717
          .
```

```
        .
        .
```

The last step is to get the numbers formatted correctly. For this—and generally to produce all kinds of precisely controlled output—we employ the `format` string method.

As a first step in understanding how `format` works, consider this code:

```
pattern = '{1} {0} dog ate {1} {2} {0} pie'
print(pattern.format('big', 'the', 'really'))
```

The string called `pattern` has numbers enclosed in braces. These are like blanks that can be filled in. When we call the `format` method, the blanks *are* filled in, using the arguments passed to `format`—in our example, the values '`big`', '`the`', and '`really`'. We think of these as being numbered starting from zero. So in the example, `{0}` will be filled in with '`big`', `{1}` will be filled in with '`the`' and `{2}` will be filled in with '`really`'. The result is that we'll see the following printed:

```
the big dog ate the really big pie
```

The main reason `format` is useful, though, is that we can exercise control over *how* items are used to fill blanks. For example, consider this code:

```
print('---{0:8.1f}---'.format(123.4567))
print('---{0:8.2f}---'.format(123.4567))
print('---{0:12.2f}---'.format(123.4567))
print('---{0:12.0f}---'.format(123.4567))
print('---{0:.2f}---'.format(123.4567))
```

In each line, the `0` inside the braces means that the blank should be filled with the first argument, the number `123.4567`. The part inside the braces after the colon (:), specifies the formatting of this number. In the first line, it is to be printed in a space eight columns wide with one digit after the decimal point. The next three lines are similar. Note though that `12.0f` specifies *no* digits after the decimal point; in this case the decimal point is omitted. Finally, as the last line shows, we can leave out the column width; if we do, the number fills as few columns as possible.

Here's the output. Match it line for line with the code and make sure you understand why each line looks as it does.

```
---  123.5---
--- 123.46---
---    123.46---
---       123---
---123.46---
```

The `f` format just illustrated, is for printing floating point numbers, numbers with a decimal part. It's just one of *many* possible formats.[1] Fortunately, we'll only need two others in addition to `f`:

- `s` means to format a string. `8s` specifies that the string should be padded with spaces on the right so that it takes up exactly eight columns. Without a number, `s` means to use the string without padding.
- `d` means to format an integer, a number without a decimal part. `5d` specifies that the integer should be padded on the left with spaces so that it takes up five columns. By itself, `d` means to use the integer as is, without padding.

Here's an example putting together everything we've covered:

```
pattern = '{3:d} Model: {1:8s} MPG: {0:4.1f}  Rate: ${2:.2f}'
print(pattern.format(224/7, 'Echo', 45.6789, 1997))
print(pattern.format(137/14, 'Escalade', 99.1234, 2005))
```

and here's the output:

```
1997 Model: Echo     MPG: 32.0  Rate: $45.68
2005 Model: Escalade MPG:  9.8  Rate: $99.12
```

Using `format`, we can now get our CPI table to look precisely as we wanted it. Here's the final code.

```
import shelve
shelf = shelve.open('cpi')
cpi = shelf['cpi']
shelf.close()

def pctIncrease(begin, end):
   return 100*(end/begin-1)

print('      Percent increase in CPI')
print('           (Jan-Jan)')
print('     1 year  5 years 10 years')

pattern1 = '{0:d} {1:6.1f}'
pattern2 = '{0:9.1f}'

for year in range(1914, 2009):
   currentCPI = cpi[year][1]
   pct = pctIncrease(cpi[year-1][1], currentCPI)
   print(pattern1.format(year, pct), end='')
   if year-5 >= 1913:
      pct = pctIncrease(cpi[year-5][1], currentCPI)
```

[1] See http://www.python.org/dev/peps/pep-3101 for details.

```
    print(pattern2.format(pct), end='')
  if year-10 >= 1913:
    pct = pctIncrease(cpi[year-10][1], currentCPI)
    print(pattern2.format(pct), end='')
  print()
```

If you run this code and look at the last column of the output, you'll quickly notice a dramatic period of deflation in the 1930s and a dramatic period of inflation ending in the mid-1980s.

2. One problem with our CPI table is that the three numbers for each year are not comparable. For example, in 2006 we find an increase of 13.2 percent over five years and an increase of 28.4 percent over ten years. Which reflects a higher inflation rate? We can't easily tell.

   A better idea would be to convert to a *per-year* rate of increase. As it turns out—we'll see how we know in a moment—a steady increase of about 2.51 percent per year yields an increase of 13.2 percent over five years and a steady increase of about 2.53 percent per year yields 28.4 percent over ten years. This makes the comparison easy: The ten-year rate is very slightly higher than the five-year rate.

   But how can we find the steady annual rate of increase that yields 13.2 percent over five years? One way would be to use mathematics. The answer we want is given by the following formula:

   $$100((1+13.2/100)^{1/5} - 1)$$

   Instead of relying on mathematical knowledge, however, we'll take an approach that's much more general. We can use it to solve not only this problem, but a wide range of others, including many in which *no one* has the mathematical knowledge to give a formulaic answer.[2]

   To start, let's write a specialized program to help us figure out the steady annual percentage increase that leads to a rise of 13.2 percent over five years.

```
def pctIncrease(begin, end):
  return 100*(end/begin-1)
```

[2] Naturally, in this particular case, it's best to use the formula if you know and understand it. Even if you do, though, learn the method we introduce. You'll need it in many applications. Just to take a simple example, you may know of a formula—the quadratic formula—for solving second-degree equations. But it can be proven that there is no such solution for fifth-degree equations. In general, given a problem with variables to the fifth power, no matter how much math you know, it is *necessary* to use our 'numerical' style of attack.

```
def increaseByPct(begin, pct):
    return begin+begin*pct/100

startValue = 175.1
while True:
    value = startValue
    pct = float(input('Enter percent: '))
    for year in range(5):
        value = increaseByPct(value, pct)
    print('Total increase: ', pctIncrease(startValue, value))
    print()
```

The first function defined here is the one we've already used. It tells us the percentage increase between two numbers.

The second function increases a number by a given percent and returns the result. The call `increaseByPct(50, 10)` returns `55`, since 50 increased by 10 percent is 55.

The main part of the program takes a percentage from the user and increases a starting value by this amount five times. Then it reports the total increase and starts the process again.

Here's a sample run to show how the program can be used.

```
Enter percent: 1.0
Total increase:  5.10100501

Enter percent: 5.0
Total increase:  27.62815625

Enter percent: 2.5
Total increase:  13.1408212891

Enter percent: 2.6
Total increase:  13.6938056761

Enter percent: 2.55
Total increase:  13.4170438657

Enter percent: 2.52
Total increase:  13.2512456622
```

If we take the CPI in January 2001—the starting value of 175.1 that we used in our program—and increase it by 1.0 percent a year for each of the five years until January 2006, the program tells us that the index will increase by a total of about 5.1 percent. This is too little—the actual increase over five years is 13.2 percent.

Next we try 5.0 percent per year, but this yields a total increase of about 27.6 percent, much too high. Now we know that the rate we're looking for is between 1.0 and 5.0 percent per year.

After a few more guesses, we narrow this range considerably. To get a total increase of 13.2 percent, we'll apparently need an annual rate of between 2.5 and 2.55 percent.

3. The approach we're using is just trial and error. Since it's so straightforward, let's automate it. Rather than getting guesses from the user, we'll just keep track of the limits of the range of possibilities and let each new guess be the midpoint of the range. If we know the number we want is between 4.0 and 6.0, we'll have the program automatically guess 5.0.

   Here's a first draft of the code:

```
def pctIncrease(begin, end):
   return 100*(end/begin-1)

def increaseByPct(begin, pct):
   return begin+begin*pct/100

def totalPct(pct):
   startValue = 175.1
   value = startValue
   for year in range(5):
      value = increaseByPct(value, pct)
   return pctIncrease(startValue, value)

lowLimit = -100
highLimit = 100

for attempt in range(20):
   guess = (lowLimit+highLimit)/2
   result = totalPct(guess)
   if result > 13.2:
      highLimit = guess
   if result < 13.2:
      lowLimit = guess

print(guess)
```

   We start by moving the computation of the total percentage increase that results from any given annual rate over five years to a function: `totalPct`. Next we set initial values for the bottom and top limits for our guesses: -100 percent and 100 percent. Then we use a `for` statement to make a series of 20 guesses. Each guess is the midpoint of the current range of possibilities. We use `totalPct` to find out what total

percentage results if this guess is used as the annual rate. If the result is above 13.2, our guess was too high, so it becomes the upper limit to use for the next guess; if the result is below 13.2, our guess was too low and it becomes the lower limit.

Either way, each new guess causes the range we're exploring to be cut in half. In this way, we very quickly home in on the answer we want. Running the code, we get an answer of 2.5106 percent, which is correct to two decimal places.

Now we'll improve our program, one step at a time. First, let's move the trial-and-error code to a function, which we'll call `goalSeek`. Rather than have key numbers like -100, 100 and 13.2 built into our new function, we'll pass them in as arguments.

```
def pctIncrease(begin, end):
   return 100*(end/begin-1)

def increaseByPct(begin, pct):
   return begin+begin*pct/100

def totalPct(pct):
   startValue = 175.1
   value = startValue
   for year in range(5):
      value = increaseByPct(value, pct)
   return pctIncrease(startValue, value)

def goalSeek(lowLimit, highLimit, target):
   for attempt in range(20):
      guess = (lowLimit+highLimit)/2
      result = totalPct(guess)
      if result > target:
         highLimit = guess
      if result < target:
         lowLimit = guess
   return guess

print(goalSeek(-100, 100, 13.2))
```

Next, rather than always making 20 guesses and hoping that narrows down the range sufficiently, let's make as many guesses as necessary to achieve a specified level of accuracy.

```
def pctIncrease(begin, end):
   return 100*(end/begin-1)

def increaseByPct(begin, pct):
   return begin+begin*pct/100
```

```
def totalPct(pct):
   startValue = 175.1
   value = startValue
   for year in range(5):
      value = increaseByPct(value, pct)
   return pctIncrease(startValue, value)

def goalSeek(lowLimit, highLimit, target, maxError=.01):

   error = maxError + 1

   while error > maxError:
      guess = (lowLimit+highLimit)/2
      result = totalPct(guess)
      error = abs(result-target)
      if result > target:
         highLimit = guess
      if result < target:
         lowLimit = guess

   return guess

print(goalSeek(-100, 100, 13.2))
print(goalSeek(-100, 100, 13.2, .000001))
```

Here we've modified the **goalSeek** function so that it takes a fourth argument, which specifies the largest error we're willing to accept. For each new guess, we check to see how far off the result is from what we want and continue so long as this error is more than the limit we've set.

A few points are noteworthy:

- When we subtract the target value from the result to get the size of the error, we will get a negative number if the target is higher. To ensure that **error** is positive, we apply the absolute value function: **abs**(13.3-13.2) and **abs**(13.2-13.3) are both equal to an error of positive .1.

- The condition of the **while** checks if the current error is still above the given limit. To make sure the condition holds the first time it is evaluated—so that we make at least one guess—we initialize **error** to a value greater than **maxError**.

- In the function header, we specify a default value of .01 for **maxError**. If we leave out this argument when we call **goalSeek**, the default value will be used. We've already seen default values in the **print** function. This is the first time we're creating our own function that provides one. In the first of the two test calls in our program, we don't specify a maximum error. The default value of .01 is used and we get the answer 2.5116. In the second call, we ask for a smaller error

and get a more accurate answer—2.5107. This produces a five-year percentage increase that differs from our target of 13.2 by no more than .000001.

Our `goalSeek` function is currently useful only for automating trial-and-error exploration of one particular function: `totalPct`. Luckily, Python treats names of functions just the same as names referring to integers, lists or any other kind of object. When we write `numbers = [3, 2, 4]`, Python assigns the name `numbers` to the specified list object. Likewise, if we write

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

Python assigns the name `example` to a **function object** with the given code. If we then write

```
apples = numbers
triple = example
```

these new names refer to the same objects as the originals and we can write

```
print(apples[2], triple(10))
```

and get 4 and 30 as output.

The upshot is that we can easily generalize `goalSeek` so that it works with any function we specify. We just need to pass in the function name. Now here's how our code looks:

```
def pctIncrease(begin, end):
  return 100*(end/begin-1)

def increaseByPct(begin, pct):
  return begin+begin*pct/100

def totalPct(pct):
  startValue = 175.1
  value = startValue
  for year in range(5):
    value = increaseByPct(value, pct)
  return pctIncrease(startValue, value)

def totalPct2(pct):
  startValue = 154.4
  value = startValue
  for year in range(10):
    value = increaseByPct(value, pct)
  return pctIncrease(startValue, value)

def goalSeek(function, lowLimit, highLimit, target, maxError=.01):

  error = maxError + 1
```

```
    while error > maxError:
        guess = (lowLimit+highLimit)/2
        result = function(guess)
        error = abs(result-target)
        if result > target:
            highLimit = guess
        if result < target:
            lowLimit = guess

    return guess

print(goalSeek(totalPct, -100, 100, 13.2, .0001))
print(goalSeek(totalPct2, -100, 100, 28.4, .0001))
```

Now that **goalSeek** can work with any function, we've defined a new one to make the ten-year calculation for 2006. This function, **totalPct2**, takes in an annual percentage increase, applies it 10 times to the CPI value for January 1996 and returns the total resulting percentage increase. We'd like this total increase to be 28.4 percent; the second call to **goalSeek** finds the corresponding annual rate.

4. Our **goalSeek** function is now very serviceable, but we have a new problem. We'd like to make five-year and ten-year calculations for every row of our CPI table. Each of these calculations will require a function along the lines of **totalPct** and **totalPct2** to be passed to **goalSeek**. Clearly, we don't want to have to create all those functions manually.

   Once again, it's lucky for us that Python treats function objects and names the same as objects and names for anything else. If we can write functions that return lists and integers, we can just as easily write functions that return functions. Here's how we apply this idea to the problem at hand:

```
def pctIncrease(begin, end):
    return 100*(end/begin-1)

def increaseByPct(begin, pct):
    return begin+begin*pct/100

def makeTotalPct(startValue, years):
    def totalPct(pct):
        value = startValue
        for year in range(years):
            value = increaseByPct(value, pct)
        return pctIncrease(startValue, value)
    return totalPct

def goalSeek(function, lowLimit, highLimit, target, maxError=.01):

    error = maxError + 1

    while error > maxError:
        guess = (lowLimit+highLimit)/2
```

```
    result = function(guess)
    error = abs(result-target)
    if result > target:
      highLimit = guess
    if result < target:
      lowLimit = guess

  return guess

totalPct = makeTotalPct(175.1, 5)
totalPct2 = makeTotalPct(154.4, 10)

print(goalSeek(totalPct, -100, 100, 13.2, .0001))
print(goalSeek(totalPct2, -100, 100, 28.4, .0001))
```

The new function, **makeTotalPct**, defines and returns functions along the line of our old **totalPct** function, but using any given starting value and number of years. In the second highlighted block of code, we show how it can be used to create the two specific functions we defined manually earlier, but of course now we can define any number of similar functions with ease.

5. That means that we're ready, finally, to take on our original problem—compiling a table of *per-year* CPI percentage increases. Here's our original code for displaying the table, modified so that calls to **goalSeek** are used to convert from five-year and ten-year increases to the steady annual rates that would produce them.

```
import shelve

def pctIncrease(begin, end):
  return 100*(end/begin-1)

def increaseByPct(begin, pct):
  return begin+begin*pct/100

def makeTotalPct(startValue, years):
  def totalPct(pct):
    value = startValue
    for year in range(years):
      value = increaseByPct(value, pct)
    return pctIncrease(startValue, value)
  return totalPct

def goalSeek(function, lowLimit, highLimit, target, maxError=.01):

  error = maxError + 1

  while error > maxError:
    guess = (lowLimit+highLimit)/2
    result = function(guess)
    error = abs(result-target)
    if result > target:
      highLimit = guess
    if result < target:
      lowLimit = guess
```

```
    return guess

shelf = shelve.open('cpi')
cpi = shelf['cpi']
shelf.close()

print('      Percent increase in CPI')
print('           (Jan-Jan)')
print('     1 year  5 years 10 years')

pattern1 = '{0:d}  {1:6.1f}'
pattern2 = '{0:9.1f}'

for year in range(1914, 2009):
    currentCPI = cpi[year][1]
    pct = pctIncrease(cpi[year-1][1], currentCPI)
    print(pattern1.format(year, pct), end='')
    if year-5 >= 1913:
        pct = pctIncrease(cpi[year-5][1], currentCPI)
        totalPct = makeTotalPct(cpi[year-5][1], 5)
        annual = goalSeek(totalPct, -100, 100, pct, .0001)
        print(pattern2.format(annual), end='')
    if year-10 >= 1913:
        pct = pctIncrease(cpi[year-10][1], currentCPI)
        totalPct = makeTotalPct(cpi[year-10][1], 10)
        annual = goalSeek(totalPct, -100, 100, pct, .0001)
        print(pattern2.format(annual), end='')
    print()
```

One last item worth attending to is the fact that we've repeated a great deal of code in the two if statements. Having two copies of a block of code means there are twice as many places for something to go wrong, twice as many lines of code to fix if something does go wrong and twice as much to modify if we decide we want to do things differently. As a design rule, therefore, it's almost always a good idea to eliminate repeated code. We can do that here by moving the duplicated part to a function.

```
import shelve

def pctIncrease(begin, end):
    return 100*(end/begin-1)

def increaseByPct(begin, pct):
    return begin+begin*pct/100

def makeTotalPct(startValue, years):
    def totalPct(pct):
        value = startValue
        for year in range(years):
            value = increaseByPct(value, pct)
        return pctIncrease(startValue, value)
    return totalPct

def goalSeek(function, lowLimit, highLimit, target, maxError=.01):
```

```
    error = maxError + 1

    while error > maxError:
        guess = (lowLimit+highLimit)/2
        result = function(guess)
        error = abs(result-target)
        if result > target:
            highLimit = guess
        if result < target:
            lowLimit = guess

    return guess

def printAnnualized(year, n):
    pattern = '{0:9.1f}'
    if year-n >= 1913:
        pct = pctIncrease(cpi[year-n][1], currentCPI)
        totalPct = makeTotalPct(cpi[year-n][1], n)
        annual = goalSeek(totalPct, -100, 100, pct, .0001)
        print(pattern.format(annual), end='')

shelf = shelve.open('cpi')
cpi = shelf['cpi']
shelf.close()

print('      Percent increase in CPI')
print('           (Jan-Jan)')
print('     1 year  5 years 10 years')

pattern = '{0:d} {1:6.1f}'

for year in range(1914, 2009):
    currentCPI = cpi[year][1]
    pct = pctIncrease(cpi[year-1][1], currentCPI)
    print(pattern.format(year, pct), end='')
    printAnnualized(year, 5)
    printAnnualized(year, 10)
    print()
```

Check the output. Now it's easy to see that, although there have been many individual years in which inflation soared to 10 or even 20 percent, *sustained* inflation has rarely even been close to that high. Over a ten-year period, average annual increases have never exceeded 9 percent. Typically, they're less than 6 percent.