Just Enough Python

version 3.5, 5 January 2024, © 2024 by Jon Seger
School of Biological Sciences, University of Utah, Salt Lake City, UT 84112

Chapter 0. Prolog: Why program, and why in Python? 1
Chapter 1. First steps 4
- ***1.1 Installing Python on your computer***
- ***1.2 Launching and navigating Python's IDLE***
- ***1.3 Python grammar 101***
- ***1.4 Working in the program editing window: Hello, World!***
- ***1.5 Random numbers and conditional execution: flipping a coin***
- ***1.6 LAB 1: some elementary (but creative) extensions***

Chapter 2. Harnessing the power of automation I: loops and lists 11
- ***2.1 The `while` loop does Kerrich's entire experiment in a split second!***
- ***2.2 Using a loop index variable to do real work***
- ***2.3 About print-formatting strings***
- ***2.4 Lists, offsets and indexes***
- ***2.5 Letting an index mean something***
- ***2.6 LAB 2: Were Wolf's dice fair?***

Chapter 3. Harnessing the power of automation II: list magic and functions 15
- ***3.1 The things lists know how to do to themselves: their methods***
- ***3.2 The `for` loop neatly steps through lists***
- ***3.3 Lists within lists, and loops within loops***
- ***3.4 Functions structure the work***
- ***3.5 LAB 3: Were Wolf's dice fair? An alternative approach using the variance***

Chapter 4. Review: 8 big ideas, the Meaning of the Universe, and where to put stuff 20

Chapter 0. Why program, and why in Python?

For thousands of years, computations of many kinds have been essential to the work of scientists, engineers, bureaucrats, community and faith leaders, farmers, explorers and business people in a wide range of fields. Yet during most of this time, actually carrying out even moderately complex computations has been difficult, slow, and highly prone to error. In other words, for most of human history, computation has been costly. Automated computing machines were long dreamed of and occasionally built, beginning especially in the 19th century. But these mechanical gizmos, made of gears, levers, dials and such, were typically designed to solve very specific problems (for example, multiplying two numbers, or calculating the orbits of planets), and they were generally more trouble than they were worth. As a consequence, people remained ignorant of the degree to which progress in their own field of endeavor (and in many others) was being limited by the cost of computation.

Finally, in the mid 20th century, advances in electronics made it possible to build fast, accurate and cheap *general-purpose* computing machines – the things we now call computers. They have profoundly changed our lives and our planet, far beyond the wildest imaginings of anyone, even the visionaries who invented modern computers. Why? Being incredibly fast, accurate, cheap and small has been important, but the real secret is that after being built, a modern computer can easily be reconfigured to solve new problems, including ones that weren't dreamed of when that particular computer was made. Because modern computers are *programmed*, they can be transformed at will into novel machines serving novel purposes, limited only by the imaginations of their programmers.

Explosive innovation on an amazing number of fronts has followed. We all enjoy the fruits of this innovation, every day. You're probably aware (as Google, Amazon and law-enforcement agencies are) that your cell phone is a computer connected to a gigantic network of other computers. But you may not be so aware of many humbler devices and activities that now have "smarts" thanks to computers embedded in them – everything from traffic lights to household appliances to your student ID card. But in every such case, the smarts are in the program that runs on the machine and gives it its personality. The computer itself is utterly stupid until programmed. And it can never be smarter than its program!

This is why you need to learn how to write programs: software can do only what it was designed to do, and life is forever posing problems whose solution demands some novel form of computation. This is most obviously true in science, but it also holds in many other academic fields including art (of all things), and in a very wide range of commercial, industrial, institutional and social enterprises, not to mention everyday life, including family life. Yet few people learn how to program any more.

We (Alan Rogers and I) decided to add a computational laboratory component to our population genetics course when we realized that the nature of research in evolutionary genetics (as in many other fields of science) is increasingly being hidden from students, as programming fades from the curriculum. Alan and I literally cannot think about problems in our field without programming, and we are typical of other practitioners of our science. How, then, could students like you possibly gain an accurate and useful understanding of fields such as ours, when you are taught in ways that shield you from exposure to an essential aspect of what practitioners actually do? After marveling at the absurdity of this state of affairs, we decided to do something about it.

Why has programming almost disappeared from the curriculum? When we were students, modern computing was still in its infancy. To get a computer to do something, you had to *tell* it what to do. That is, you had to write a program of some kind. So we and our peers all learned to program, because we wanted to use computers in our research. But during our careers, software has become graphical and "user-friendly". These developments have allowed everyone to easily use computers to write letters, communicate by e-mail, make impressive graphic presentations, organize data in spreadsheets, surf the web, chat with chat-bots, and more. However, there's a little-recognized down side to these developments as well: they have inverted the relationship between human beings and computers.

Forty years ago, people told computers what to do. Today, computers tell people what to do. Computers are easy to use today mainly because software now presents the user with a small number of choices. It typically says to us, "Click one of these two options." We almost never get a chance to tell it what choices to present in the first place. Who works for whom here? And who is doing the thinking? Mostly this is not a problem, because the relevant choices can be anticipated when we're engaged in a well defined activity like writing a letter (or an essay on computing, like this one). These predictable,

stereotyped activities can be made very efficient by reducing them to fill-in-the-blanks exercises. But what if we need to solve a problem that hasn't already been solved by the programmers at Microsoft or Google? *What if we need to have a new thought?* Because modern software presents small numbers of predetermined choices, users inevitably try to recast their problems as ones that the software they know (*Excel,* or ChatGPT, or whatever) already knows how to solve. Sometimes that works well enough, but innovation tends to be stifled when the questions we feel entitled to ask are limited to ones just like those that have already been asked so often that they are now enshrined in widely available software.

The solution to this problem is not to reject modern software, which is very good at what it does, but instead to re-learn the art of training the computer to do what *we* think it should do, when we have our own ideas as to what we would like to see happen. Fortunately, the tools available for doing this are vastly better than they were when Alan and I were young. There's now a great diversity of programming languages and resources tailored to different kinds of tasks. Among these modern languages, Python is rapidly gaining popularity because it is very easy to learn and it runs on all kinds of computers (Windows, Mac, Linux, *etc.*). It lets beginners start doing real work in almost no time, yet also has the kinds of advanced features that professionals need to attack complex, real-world tasks. And it's completely free — you just download it from the web. It's developed and maintained by a world-wide community of volunteer programmers that welcomes anyone willing and able to contribute.

So why can't an evolutionary geneticist think without programming? Actually, we do sometimes think without programming, but our trains of thought often raise questions that need to be answered before we can take the next step. And often the most direct way to get the answer is to do a computation of a particular kind. Sometimes the question is a theoretical "what-if" (for example, what level of genetic variation should we expect to find in a population of a certain size that had been growing at a certain rate for a certain number of generations). And sometimes the question is an empirical "what-is" (for example, how do levels of variation change along a certain human chromosome as we get closer to a gene with mutations that cause a certain disease).

To answer the "what-if" questions we write programs that *model* (*i.e.,* mimic) the evolutionary processes of interest. Computers are invaluable here because they often let us answer a question by doing something that is conceptually very simple, but so tedious that it could never be done by hand, at least not in practice. To answer the "what-is" questions we write programs that *digest* large amounts of genetic or other data and summarize them in a form that directly answers the particular question we are asking. Here, too, the principles being applied are usually very simple, but the amounts of data are so large that it would not be practical to carry out the analysis by hand. In cases of both kinds (what-if and what-is), programming lets us get the answers quickly, while the question is still burning in our minds, and before we've forgotten why we asked it in the first place. This greatly facilitates the process of "connecting the dots" which is vital in science (as in other fields, and in real life).

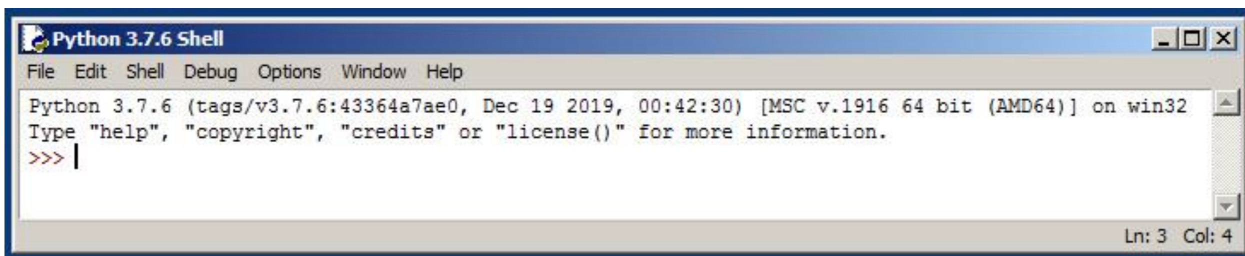
You are about to start learning an entirely new skill and way of thinking. It may seem rather strange at first, but please don't panic or despair. It's simpler and easier than it looks, and once you get the hang of it, a lot of fun in addition to being endlessly useful. We'll start with an exercise of the what-if variety, in which we build a model of John Kerrich's heroic coin-flipping experiment. (This experiment is described in the companion to this tutorial, called *Just Enough Probability*.)

Chapter 1. First steps

1.1 Installing Python on your computer. If you're going to use your own computer for these exercises, then download an appropriate Python 3 installer (for Windows, Mac, or whatever) from **`python.org`** and run it. This is easy and doesn't take long. As of December 2023, the latest stable production version is **Python 3.12.1**. The very latest versions can be buggy, so avoid them, and also ***don't install Python 2*** because it is a slightly different "dialect" of the language, and is no longer being maintained.

1.2 Launching Python's IDLE. There are several ways to work with Python, but the easiest is to use its built-in "Integrated DeveLopment Environment" called IDLE, which provides an intelligent text editor for writing Python code (the ***editing window***) and a window for running programs and interacting directly with the Python interpreter (the ***Python "Shell"***).

To launch IDLE just double-click on its icon (or a shortcut that you have placed on the desktop or the taskbar or wherever). When IDLE launches it opens the ***Python Shell window***:



The *Shell* provides a *command-line* interface to the Python *interpreter*, which is itself a program that *reads* Python *statements* and *executes* them. Note the Python *prompt* (`">>>"`). The cursor is sitting to the right of the prompt, waiting for you to type a Python statement. OK, here we go. Enter (type) the following line, exactly as printed below, including the quotation marks, and then a "carriage return" (that is, tap the Enter key to terminate the line).

```
print("Hello, World!")
```

You should now see something like this:

```
>>> print("Hello, World!")
Hello, World!
>>> |
```

To save paper, we show just the relevant part of the Python Shell window, as text, rather than the whole window, as a graphic. To the right of the upper prompt is what you entered. Below it is what Python did in response to your statement: it printed the characters between the quote marks, but not the quote marks themselves. Then it printed another prompt, with the cursor (a blinking vertical bar) to its right.

What you just did was use the Python interpreter in what's called *interactive mode*. Whatever you enter at the >>> prompt, Python will attempt to execute it immediately. And if it makes sense to print the result, then Python will print it even if you didn't ask to have it printed. Thus the Shell allows you to try out Python statements and get immediate feedback (the result, or else an error message if what you entered wasn't a legal Python statement). This is how to learn Python – just try stuff out, fearlessly!

1.3 Python grammar 101. The Shell also lets you use Python as a wickedly powerful and convenient calculator. To see why, and to learn some important first lessons in what might be called "the grammar of Python", enter the following lines in succession at the prompt, and note carefully what happens, or doesn't happen. (Type each line exactly as printed here, and then the Enter key.)

```
a = 6
b = 7
a
b
a*b
b = "7"
b
a*b
s = "Spam "
a*s
b*s
```

(For the line `s = "Spam "`: <Please enter the ***space*** after "m", but ***do not*** enter this note in angle brackets!>)

Here `a` always refers to the integer 6, but `b` refers first to the integer 7 and later to the character (string) `7`, which is a different kind of object. Python tells us this by enclosing it in single quotes as `'7'`. And ***we*** type quotes to tell Python that we want it to treat something as a string. Single and double quotes can both be used to enclose strings of characters in Python, as long as they're used in pairs (`'Spam '` or `"Spam "`, but not `'Spam "`).

When `a` is the ***number*** 6 and `b` is the ***string*** `"7"`, the operation `a*b` yields the *string* `"777777"` (not `42`, and not the *number* 777777). Similarly, `a*s` is `'Spam Spam Spam Spam Spam Spam '`, but `b*s` is an error, because you can't multiply one string by another.

OK, let's turn `b` back into a number. Type:

```
b = 7.0
b*s
```

Again an error, because `b` now refers to a *floating-point number* (called a "float" in computer jargon), and Python allows strings to be multiplied only by integers. Finally, try this:

```
a*b
```

This works, because integers and floats can be multiplied together in Python, just as in real life. But in Python the result is an object of type *float*, even if the number has no fractional part. 42.0 has the same value as 42, but to Python it's a different object, of a different type, and as a consequence, it has different properties. And it's *not* the Meaning of the Universe! (Yes, Python is named after the British comedy group Monty Python, and Douglas Adams of the *Hitchhiker's Guide* worked with them.)

And speaking of the Galaxy, astrophysicists estimate that the number of elementary particles with mass in the Universe is roughly 10^{80}. Let's see that number. In Python, `**` is the exponentiation operator, so 10^{80} can be represented as `10**80`. Back at the Shell prompt, type

```
U = 10**80
print(U)
```

That's supposed to be 1 followed by 80 zeros, right? Is it? Let Python do the counting. U is a variable (that is, a name) that "points to" an integer "object" that represents the ***value*** 10^{80}. Python represents all integers – small and large – in a unique scheme that represents their values exactly. (Almost no other programming languages can handle integers anywhere near the size of U.) When we say `print(U)`, Python prints a string representation of the value U, in our usual base-10 number system. We can also use the function **`str()`** ["string"] to assign that string representation of U to a variable. So type

```
sU = str(U)
sU
len(sU)
```

Does that make sense? The length of the string representation of `U` is 81 characters. One of those is "`1`". The others must be zeros. Also note that everything in Python is case-sensitive: "`u`" is not "`U`".

What have we learned here, about the grammar and logic of Python?
(1) The numbers and other things we represent in a program are thought of as ***objects***;
(2) they have ***types*** corresponding to the kinds of things they represent (integers, floating-point numbers, strings of printable characters, *etc.*); and
(3) ***variables*** are arbitrary names that refer or "point" to those objects in ***statements*** that we write to tell Python what ***operations*** we want it to carry out on the ***values*** of the objects.

In Python, variables are created when we ***assign*** some value to them. The value and the variable name are completely different things, maintained in different parts of the computer's memory (and right now there is no "`u`"). When you say `print(U)`, Python prints the *value* of the *object* to which `U` points (in the example above, the integer 10**80). You'll see the same on-screen representation if you say `print(sU)`, but the value of `sU` is *not* a number, it's a long string of printable characters. Python tried to alert you to this fact when you just typed `sU` at the Shell prompt, by enclosing its representation of `sU` in single quote marks. In general, Python tries to tell you what kind of object a variable represents, and you should train yourself to notice those hints. For example, the floating-point number 42.0 will always be printed with the ".0", never as "42", unless you force Python to print it that way.

As far as we are aware, no other programming language distinguishes so completely and consistently between *names* (variables) and *the things names refer to* (objects and their values). Any variable can refer to any object, including objects of *different types* at different points in the same program. You illustrated this flexibility above, where `b` pointed first to the integer 7, then to the string "`7`", and then to the floating-point number 7.0. In most other programming languages, that would not be allowed.

Now, with these preliminaries behind us, let's write and run some actual programs!

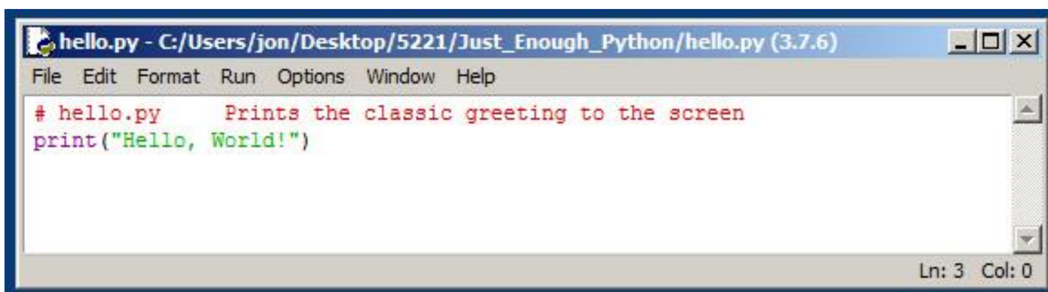
1.4 Working in the program editing window: Hello, World! A computer program is a special kind of *script*, like the script for a play or a movie that creates a bunch of actors and describes the actions they perform, separately and in various combinations with each other. Here we will learn the mechanics involved in making such a script for Python. In the culture of computing, there's an ancient tradition that a beginner's first program in any new language simply writes "Hello, World!" to the screen. So

let's do that one first. Then in the next section we'll make a program that flips a coin, as our first step toward modeling Kerrich's famous wartime entertainment.

The work you did previously in the Python Shell will disappear when you exit Python. To create a script that can be saved and run again at some arbitrary time in the future, you need to open IDLE's *program editing window*. Click File > New Window in the Python Shell menu system, or use the keyboard shortcut (command-N on the Mac, control-N in Windows). As you would expect, this opens a new window, but it's not another Python Shell. Instead, it's a simple but intelligent text editor that knows how to communicate with the Python Shell. There's a cursor blinking at the beginning of the first line. Enter the "Hello, World!" program, which consists of these two lines:

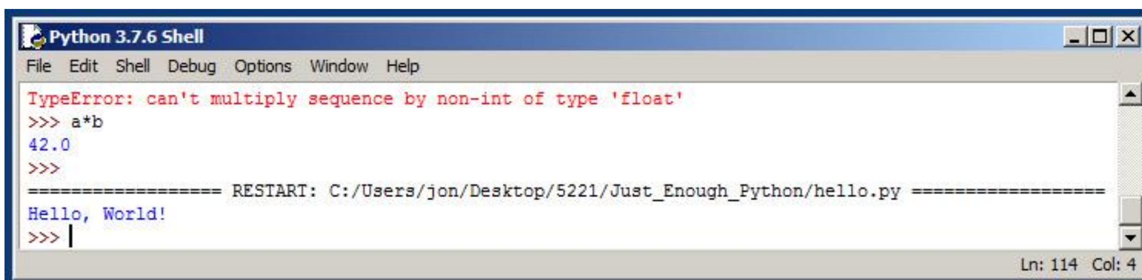
```
# hello.py   Prints the classic greeting to the screen
print("Hello, World!")
```

As you enter text, you'll see the various words and expressions turn different colors, which are designed to help you easily distinguish the major kinds of program elements (for example, *comment* lines turn red). Now save your program to the Desktop (or some other appropriate location) using File > Save As. Give it the name "hello.py". The result should look like this:



Note that the title bar now shows the name of your program. Python "modules", including programs, normally have names that end with the extension ".py", so that IDLE and other parts of the Python package can recognize them. If you look on the desktop (or wherever you saved it) you'll find your program sporting the Python icon. Inside is just a two-line text file, which you could view with any word processor or plain text editor, and containing exactly the text you see in IDLE's editing window (but without the colors). Now let's run it!

Click Run > Run Module, or use the keyboard shortcut which is F5 (function-key 5). The focus will switch from the editing window to the Python Shell, and in a second you'll see some new text appear in that window. The result should look something like this:



When we left the Shell after the Python grammar lesson, it showed a >>> prompt right after 42.0, which was its report on the type and value of `a*b`. Running hello.py then caused a Shell "restart", erasing all of the variables that we had previously created and the objects to which they referred. Then the Shell read and executed the contents of hello.py (those two simple lines).

All text to the right of a number sign (#) is treated as a comment to be ignored. The second line of your program causes "Hello, World!" to be printed to the screen, as you now understand from the previous interactive exercises. By putting the statement in a script, and then running the script, you have *automated* what you previously did manually. This may not seem like a great advance, but it is. That's because you can put *any number* of statements in a program, which will then execute them *in an order that can itself be programmed*. Thus even a *few* statements may carry out a complex and lengthy calculation. ***That's the magic and the power of programming.***

You now understand why the quotation marks disappeared: the `print()` function displays the *value* of its argument, which in this case is the string of characters *between* the quotation marks that tell Python to treat `Hello, World!` as a string – that is, as an object of the type *string of characters*.

1.5 Random numbers and conditional execution: flipping a coin. How can we write a program to model the process of flipping a coin? The answer is to simplify or *abstract* the process down to just that aspect that we care about *for present purposes*. In this case, what we care about is whether a given flip comes up heads or tails. We don't care *which dead president's head* is on the coin, what *year* the coin was minted, or anything else about it – just whether we get "heads" or "tails". For now we will assume that heads and tails occur with probability 0.5 (that is, that the coin is "fair").

OK, from IDLE's Python Shell, open a New Window (you may first want to close `hello.py`). Now enter the following program, and save it as `coin1.py`.

```
# coin1.py   simulates one "spin" (flip) of a fair coin
# LOAD THE RANDOM NUMBER GENERATOR
from random import random
# GET A RANDOM NUMBER
p = random()
# IF LESS THAN 0.5, CALL HEADS
if p < 0.5:
   print("heads!")
else:
   print("tails!")
```

This program is very similar to the one in the "answers to exercises" section of *Just Enough Probability*. It has several elements that we didn't see in `hello.py`. The first non-comment line loads a Python *module* that generates random numbers. That line translates into English as "from the `random.py` module, load into memory the `random()` function and make it available to this program". The `random` module is a big file full of Python code that is installed with the other pieces of Python, so it's always available to be loaded. There are other such modules, including one that provides mathematical functions to do things like taking the square root of a number. Many other modules have been written

by programmers (both amateurs and professionals) and posted on the web, where they can be downloaded to augment the "native" Python installation and provide additional functions that can be imported to programs written by users like us. This sounds impressive, but in fact it's not difficult to write reusable modules that extend the powers of Python. For now, however, we just need to know how to import functions such as `random()` and `sqrt()` from the standard modules such as `random.py` and `math.py`.

The second non-comment line invokes or *calls* the `random()` function, whose workings we do not need to understand. (That's the whole point of packing it into a module!) We just need to know that it returns a number we *can't predict* except to say that it is equally likely to be anywhere between 0 and 1. Our program stores this *uniformly distributed random number* as a floating-point object and associates it with the variable `p` which can then be used to read the number's value.

The last four non-comment lines form a *conditional execution block* that prints either "heads" or "tails", depending on the value of `p`. Carefully note the forms of these *compound statements*. The first part of each compound statement (beginning at the left-hand margin) *tests a condition*, for example, `if p < 0.5`, and ends with a *colon*. The second part, on the line below, is *indented*. That line is executed *only* if the first line is *true*. So if `p` (the random number) is less than one half, then Python prints "`heads`" to the screen. But if `p` (the random number) is *not* less than one half, then the other test ("`else:`") is true and "`tails`" is printed. This *algorithm* for choosing an action based on a variable condition takes uniformly distributed random numbers and turns them into simulated heads and tails. It's the heart of the model, and a perfect example of how computation can be used to mimic a natural physical process.

Important: Indentation defines program blocks! Python uses indentation to set off statements (or blocks of statements) that should be executed conditionally (as here) or repeatedly (as we will see shortly). The syntax is always the same, and it must be followed *exactly*. The general form is

```
[if X is true]:
    [then do Y]
    [and Z]
    [and so on ...]
```

IDLE helps you follow this style, by inserting indentations automatically following a conditional statement terminated with a colon. ***The indentations should be done to the same depth, and by the same means (either with spaces, or with tabs) for all of the statements in a block or a set of logically connected blocks.*** (IDLE uses 4 spaces. You will have a much better life if you follow its lead.)

When you have saved the program, run it. Then run it again, and again, and again. You should get a seemingly random sequence of "heads" and "tails".

How does it work? Every time it is called, the `random()` function generates a *different* number between 0 and 1, with an equal chance of being anywhere within the interval. Our program exploits the fact that in 50% of the cases this number will be less than 0.5, and thus that in the other 50% of the cases it will be greater than 0.5. If we call "heads" when the number is less than 0.5 and "tails" otherwise, then we will get 50% heads and 50% tails, in a random sequence. Note that we could equally well call

tails when the number was below 0.5 and heads otherwise. Or we could call heads when the number was between 0.25 and 0.75, and tails when below 0.25 or above 0.75. As long as each "side" of our virtual "coin" corresponds to *half of the total interval between 0 and 1*, we should get heads half of the time, and tails the other half of the time. ***And no one could distinguish the output of our program from a record of the flips of an actual coin!*** (A fair coin, anyway.)

1.6 LAB 1: some elementary (but creative) extensions. Python's "if-else" test includes a third condition, "elif", which stands for "else if". Here's the code for a three-way test to make "heads" correspond to `p` between 0.25 and 0.75.

```
if p < 0.25:
    print("tails!")
elif p < 0.75:
    print("heads!")
else:
    print("tails!")
```

Make a modified version of the program (call it `coin2.py`) that uses this three-way test to decide which trials turn up heads and which turn up tails. Run each program 20 times and record the sequences of heads and tails you get. Are the totals close to 50-50 in both cases? Do the sequences seem random? Now, invent a *four-way* division of the range between 0 and 1 into intervals where the first and third total to 0.5, as do the second and fourth. Just for fun, don't do the obvious thing of dividing the interval at 0.25, 0.5 and 0.75. Write a four-way conditional test that maps your intervals onto "heads" and "tails", and use it to make yet another version of the program called `coin3.py`. Run this program 20 times and record the sequence of heads and tails. If you've done everything right, it too should look for all practical purposes like 20 flips of a fair coin. Does it?

Now, for your last performance, turn `coin3.py` into a program called `die1.py` that simulates a *fair die*, which when rolled is *equally* likely to show the sides with 1, 2, 3, 4, 5 or 6 spots. This will prepare you for the next lab, where we will investigate the fairness of Rudolf Wolf's red and white dice (see *Just Enough Probability*, page 5).

Your written report for Lab 1 should include your coin3.py and die1.py programs, and the outputs generated by running them 20 consecutive times. You can capture the outputs from the Python Shell window by copy-and-paste to a word processor.

Chapter 2. Harnessing the power of automation I: loops and lists

2.1 The* `while` *loop does Kerrich's entire experiment in a split second! In section 1.5 you made a program that simulates flipping a coin, once. Very clever, but to run it takes as long as to flip a real coin, so in its present form this program is not very useful. However, we can easily modify it to simulate as many flips as we like, and doing so shows why programs can be such powerful tools.

The trick is to place the code for some action inside a *loop* that executes that code *repeatedly* until a desired condition is satisfied. In our case this would involve flipping the coin some number of times while keeping track of the resulting numbers of heads and tails. Suppose we want to flip the coin 20 times (as in the previous exercises). Like most other programming languages, Python provides a form of the `while` loop which executes a block of code as long as a specified condition remains true. The syntax of `while` is much like that of the `if` conditional-execution statement that you already know, except that the `while` block *continues* to execute until the test *fails*:

```
while <test is true>:
   <do this>
```

Thus, to collect 20 flips of the coin we could write:

```
i = heads = 0
while i < 20:
   p = random()
   i += 1
   if p < 0.5:
      heads += 1
```

A loop: The three lines indented one level to the right of the `while` statement will be executed in order, again and again, as long as `i < 20`. On each of these occasions, the fourth line (indented one level to the right of `if p < 0.5:`) will execute if `p` (the random number) is less than ½.

This code fragment introduces several important ideas. First, more than one variable can be initialized on a single line (here `i` is set equal to `heads` after `heads` is set equal to `0`). Second, the value of a variable can be incremented using the *extended assignment* operator "+=". Thus "`i += 1`" is equivalent to "`i = i + 1`". Third, conditional execution blocks can be *nested* inside others to create *hierarchical* program structures. Here the `while` block will execute 20 times (until the value of `i` reaches 20), while the `if` block nested inside it will execute *some fraction* of those times (50% on average, but the exact number will vary depending on the output of `random()`). Fourth, we don't need to keep track of `tails`, because tails = <total flips> - heads.

Here's a complete program that flips the coin 10,000 times and reports the result. Please write one similar to this and run it a few times. About how many seconds does it take to run? About how many seconds *per flip* is that? How does the proportion of heads vary? How do these proportions compare to what you expect, and to Kerrich's data for his real coin?

```
# coin2.py   simulates Kerrich's entire experiment
from random import random
i = heads = 0
while i < 10000:
   p = random()
   i += 1
   if p < 0.5:
      heads += 1
print(heads, heads/10000)
```

Beware the infinite loop! Note that if we had forgotten to increment `i` by 1 during each iteration of the loop, then the test at the top would remain true forever (`i` would always be 0, which is less than 10000). Every beginning programmer accidentally creates such "infinite loops" fairly often, and even seasoned experts do it occasionally. Fortunately, you do not need to unplug the computer to recover from such disasters. "Control-C" (simultaneously pressing the *Crtl* key and the letter *C*) is understood by most operating systems and by much software (including Python) as a polite but urgent request to please stop this runaway program and give me back my computer!

2.2 Using a loop index variable to do real work. In the previous example, we didn't do anything interesting with `i` (the loop "counter" or "index variable"). We just used it to keep track of the number of times we had flipped our imaginary coin, so that we could know when to stop (on reaching a total of 10,000 flips). But often it makes sense to use the *value* of the index in a calculation where it represents some quantity of importance to our problem. For example, suppose we wanted to know the sum of all the integers from 9 to 90. We could figure this out with just a few simple lines of Python code:

```
total = 0
i = 9
while i <= 90:
    total += i
    i += 1
print("sum = %d" % (total))
```

All we have to do is add the current value of `i` to the current value of `total`, for every integer value of `i` between 9 and 90. The answer is 4059. (There's an algebraic solution to this problem, but if you don't already know it, then the brain-dead programming solution will get the answer more quickly.)

2.3 About print-formatting strings. In computerese, a "string" is a sequence of characters, such as this sentence. We often want to carefully control the way we print numbers or words, for example when we want them to line up in neat columns. If x is the floating-point value 10.0/3.0 and you say "print(x)", Python will print `3.333333333`… to many more decimal places than you want. But if you say

```
print("%5.3f" % (x))
```

you'll get `3.333`. The string in double-quotes that appears as the first argument to the print() function is a "formatting string" or "format specifier" that describes a rule to be followed in printing the value in the second argument, which is the number x. `"%5.3f"` means "show a floating-point value 5 characters wide, with 3 of them to the right of the decimal point." You can expand the formatting string to handle several values and different kinds of values. For example, if y=100 and z=300, then

```
print("%4d/%-4d =  %6.4f" % (y,z,y/z))
```

causes

```
 100/300  =  0.3333
```

to appear on the screen. Note that there's a space to the left of "`100`" and one to the right of "`300`", which is left-justified within its 4-character space for decimal digits by the format specifier "`%-4d`". The "/" and the "=" were just printed, as is, because they aren't legal format-specifying formulas.

2.4 Lists, offsets and indexes. Often it is more convenient to work with *arrays* of variables than with individual variables such as `i`, `heads`, `p and total`. This will be true, for example, in an exercise to be introduced shortly where we will collect data on simulated rolls of a die. (The outcomes will be integer values between 1 and 6, and it will be efficient to collect them in an array.) In Python, arrays are represented as *lists* which hold *ordered collections* of other objects which may be numbers, strings, or *other lists* of such objects. Let's explore a few of the properties of lists interactively in the Python Shell. Please move the focus to the Shell by clicking inside it. Then, to make a list `x` containing three floating-point numbers, enter:

```
>>> x = [1.3, 2.4, -0.8]
```

To examine this thing enter:

```
>>> print(x)
>>> print(x[0])
>>> print(x[2])
>>> print(x[3])
```
<This generates an error. Why?>

Note that `x` is the ***list*** itself (denoted by its outer square brackets and elements separated by commas), while `x[0]` and `x[2]` are ***numbers*** that reside in the list.

The *members* or *elements* of a list are referred to by their *positions* in the list, which we think of as *offsets* from the first element. Square brackets are used in this context to indicate that the number inside is the desired offset, relative to the first element. Thus `x[0]` refers to the first element itself (the one with zero offset), and `x[2]` is the last element (the third one). Here's a picture:

	[0]	[1]	[2]
x	1.3	2.4	-0.8

The advantage of referring to these three numbers as `x[0]`, `x[1]` and `x[2]`, rather than the simpler `x`, `y` and `z`, is that the offsets can be values stored in *variables*, rather than being written out explicitly as above. To see how this works, enter the following lines at the Shell prompt:

```
>>> i = 0
>>> print(x[i])
>>> i = 1
>>> print(x[i])
```

Notice that you used *exactly the same statement* (`print(x[i])`) to refer to *two different objects* (`x[0]` and `x[1]`), depending on the value of `i`. Thus we can refer to the different members of a list by *calculating* their offsets, or in the simplest case, by using a loop index variable to "address" or "point to" a succession of adjacent members of the list. Any valid numerical variable can be used as an index, but `i` is very popular in this role because of the obvious relationship between array indexes and traditional mathematical subscripts (for example, x_i). And `i` is also the first letter of "index".

2.5 Letting an index mean something. Often the position of an element in an array is meaningless except as the "address" where we can find it – that is, as a numerical part of its name. But sometimes it makes sense to associate array offsets with the *values* of data. To illustrate this idea, let's write a program that records the rolls of a simulated die. At the end of the first lab, you simulated a die by

drawing a random number *p* and then using a series of if/elif/else statements to identify which sixth (1/6, covering 16.67%) of the unit interval contained the random number. That's the most *conceptually straightforward* way to simulate a random process with six equally likely outcomes, but it's *not the simplest* algorithm! What if we multiplied each `random()` number by 6? Then whenever p was smaller than 1/6 (0.166666...), $6p$ would be smaller than 1, and when p was between 1/6 and 2/6, then $6p$ would be between 1 and 2, and so on. If we took the *integer* parts of these products, we would get the integers 0, 1, 2, 3, 4 and 5, *with equal probability*. To get the sides of a fair die, then, all we would have to do is add 1. At the prompt enter:

```
>>> from random import random
>>> print(1 + int(6.0*random()))
```

Here one line of code accomplishes what required more than 10 lines (and a lot of fussy indenting) when we did it with `if/elif/else` conditional execution statements. This illustrates the general principle that there are usually many different ways to do something in Python or any other programming language, and that some of these ways are usually simpler and/or more efficient than others. Now, to collect 1000 rolls of our simulated die, all we need to do is use the *value* of each roll to refer to one member of a list with elements 1 through 6, where we will keep a running tally of the rolls. Here's a little program that does this. Write and run it, please!

```
# one_die.py
from random import random
rolls = [0,0,0,0,0,0,0]
i = 0
print(rolls)
while i < 1000:
    spots = 1 + int(6.0*random())
    rolls[spots] += 1
    i += 1
print(rolls)
```

The `rolls` list is set up with 7 elements (integers initialized to 0), because indexing begins at 0 in Python and we want to use `rolls[6]` (the seventh element) to accumulate the number of rolls where the simulated die shows 6 spots. After the loop has executed 1000 times, we will have six numbers that sum to 1000 (each averaging about 167) in `rolls[1]` through `rolls[6]`. To see this, add the following lines to the end of `one_die.py` and run it again.

```
i = 1
while i < 7:
    print i,rolls[i]
    i += 1
```

This shows explicitly how we have associated a variable of interest (the number of spots showing on the die) with an index, and used it to control which member of the `rolls` list we will increment during the "experiment", and which member of the list we will print at the end.

2.6 LAB 2: Were Wolf's dice fair? A fair die should show each of its faces 1/6 of the time, on average. In 20,000 rolls, that would be 3333.3 times. Wolf's white die showed three and four spots fewer than 2900 times each, and it showed six more than 3900 times. The red die also showed four fewer than 3000 times. These numbers seem suspiciously far from the expected values, but are they extreme enough to be evidence that the dice are not actually fair? After all, even a perfect die will give numbers that differ from the expectations.

	White						
Red	1	2	3	4	5	6	sum
1	547	587	500	462	621	690	3407
2	609	655	497	535	651	684	3631
3	514	540	468	438	587	629	3176
4	462	507	414	413	509	611	2916
5	551	562	499	506	658	672	3448
6	563	598	519	487	609	646	3422
sum:	3246	3449	2897	2841	3635	3932	20000

Table 3: The results of 20,000 throws with two dice (Wolf 1850, cited in [1])

Modify your **`one_die.py`** program to collect the outcomes of 20,000 consecutive rolls, as in Wolf's experiment. Run it 20 times, keeping track of the highest and lowest totals you obtain in each run. Do you ever get as few threes or fours as Wolf did, or as many sixes? Is that the right question to be asking? How should we compare the simulations to the data? Do you think your results make a case for the fairness or unfairness of Wolf's dice? Later in the course we'll revisit this approach to testing statistical hypotheses, which is very powerful. Your lab report just needs to include your program, your results, and your brief but thoughtful answers to these questions.

Chapter 3. Harnessing the power of automation II: list "methods" and functions

3.1 The things lists know how to do to themselves: their methods. Lists are usually used to group logically related elements of the same type, but in Python this is just a convention, not a requirement. At the shell prompt type `x = [3, 2, 1]  <return>`, then `print(x)  <return>`.

Now type `x[1] = 'spam'`, then `print(x)`. (Or just type `x` , which will also print `x`.)

Lists give Python great power because they are so flexible, and also because they have a number of extremely useful *methods* that they know how to apply to their contents. At the prompt enter:

```
>>> x.append('eggs')
>>> print(x)
>>> x.sort()
>>> x
>>> x.reverse()
>>> print(x)
```

The `append` method "grows" a list by adding the element between parentheses (its "argument") to the end of the list. The `sort` method arranges the list in ascending order (alphabetical and/or numerical), and `reverse` does just what it says. There are many other list methods, which you can read about in the Python Standard Library Reference which is available from the Python Shell under the "Help" drop-down menu. Strings also have some list-like properties and a rich set of methods (see the **Review** chapter, below). You can exercise some useful string methods by typing these lines at the prompt:

```
>>> s = "genome"
>>> print(s)
>>> print(s.upper())
```

```
>>> print(s[2])
>>> print(s.find("ome"))
>>> print(s.replace("ome","ts"))
>>> print(s)
```

Lists and strings are Python objects belonging to the *sequence* category, which means that their elements are *ordered* and can be referred to with indexing. But lists and strings differ in many other ways, including that lists are *mutable* but strings are *immutable*. When we sorted the list `x` (`x.sort()`), it remained sorted. But after everything we did to string `s` above, it is still the same as when we first defined it. If we want to *change* a string, then we have to *assign* the result of some transformation of it to a new string (which may, optionally, have the same name as the old string). Enter these lines:

```
>>> s = s.replace("ome","ts")
>>> print(s)
```

In computerese, the string methods *return* the result of their action as a *value* that can be printed or assigned to a variable. But the methods do *not* modify the *object* referred to by the *name* of the string (`s` in this case). By contrast, many of the list methods work differently, modifying the list "in place".

3.2 The `for` loop neatly steps through lists. Now we're ready to meet the `for` loop. It's more sophisticated than `while`. Instead of repeatedly testing some condition in a head-banging sort of way, `for` works methodically through a list – any list! At the Python Shell prompt, try this:

```
>>> my_data = [1,9,2,8,3,7,4,6,5]
>>> print(my_data)
>>> for x in my_data:
        print(x)
```

The third line introduces a ***new idea***: for each of the elements in the list called `my_data`, in order, assign the value of that element to the variable `x`, and then execute the indented block of statements. Here the indented block is trivial: `print(x)` (the value of the current element of `my_data`). But we can do many things with this device, which allows us to examine and act on the elements of any list. For example, we can drive a `for` loop by stepping through a sequence of integers generated by the `range` function. At the Python Shell prompt, try this:

```
>>> print(range(10))
>>> for i in range(10):
        print(i)
```

`range()` is a function that returns a list of integers. With the argument 10, it returns a list filled with the integers from 0 to 9 (the first 10 integers). In other words, the values of the elements of the list that `range` creates are set equal to the *offsets* of those values. To see this, try:

```
>>> x = range(10)
>>> print(x)
>>> print(x[4])
>>> for i in x:
        print(i)
```

This is why `for i in range(n):` is such a popular way to loop exactly **n** times through a block of code.

3.3 Lists within lists, and loops within loops. This is where programming starts to become really interesting and powerful. First, the elements of a list can be other lists, and we can refer to the elements of the "inner" lists with *secondary subscripts*.

```
>>> x = [[1,2,3],[4,5,6],[7,8,9]]
>>> print(x[2][1])
```

Second, in a similar way an "outer" loop can run an "inner" loop. Please type this at the Shell prompt, then another carriage return (Enter) to make it run.

```
>>> for i in ['eggs','spam']:
        for j in range(1,3):
            print(i,j)
```

3.4 Functions structure the work. Believe it or not, you've now met most of the great ideas or "secrets" of programming. And here's the last one: stuffing stuff into *functions*. We've already used several functions including `print()`, `random()` and `range()`, without explaining exactly what they are, how they work, or the important fact that *you can write your own*. A function's name can be thought of as a "handle" that refers to a free-standing block of code that is located somewhere else. In the jargon of computing, your program "calls" a function by saying its name. Often the function "returns" a value that it has computed from a supplied input or "argument" value. This is always true for computational functions that implement traditional mathematical functions. For example, `y = log(x)` causes the natural logarithm of `x` to be assigned to `y`, just as you would hope.

`range()` returns a list of integers. With one argument [as in `range(10)`], it begins the list at 0, and then steps through consecutive integers to one less than the argument (to 9 in this case). With two arguments, it begins the sequence at the first argument instead of at zero. And with three arguments it takes steps of the size specified by the third argument, which can be negative. Thus `range(9,0,-2)` returns `[9, 7, 5, 3, 1]`. Try this at the Shell prompt, but with arguments of your own choosing!

`random()` doesn't normally take an argument; it just returns an unpredictable floating-point number between 0.0 and 1.0. Another very useful function is `sum()`, which adds the numbers in a list. Thus `sum(range(9,91))` is 4059. That's the sum of the integers from 9 through 90, which we calculated using six lines of code and a `while` loop in section 2.2, on page 12, above.

Some functions such as `range()` and `sum()` are built into the core of the Python language, so they are always available. Others must be imported from a module [for example, the `random()` function is imported from the module `random.py` which also contains other random-number generators, and the `log()` function is imported from `math.py` which contains `exp()`, `sqrt()` and many others]. You can also write your own functions, with code that resides inside the Python program that will call the function. This is often a useful thing to do, because it allows you to write an algorithm once and then use it at various different places in your program, just by calling it.

For example, in this course we are obsessed with the variance. It's such a simple concept, but often tedious to calculate. Python provides `sum(my_list)`, but it does not provide `mean(my_list)` or

`var(my_list)`. So let's do it! A function that returned the variance of the numbers in a list is a *tool* that we could use over and over, in many different situations. How could we calculate the variance of a list of numbers? One straightforward approach is to use the definition of the variance: the expected value of the squared deviations of the numbers from their mean. Thus we could first sum the list and divide by the number of elements in it (returned by the built-in function `len()`), then sum the squared deviations from the mean and again divide by the number of elements.

```
def var(x_vector):                   # x_vector is the list of numbers in our sample
   sum_x = float(sum(x_vector))          # NOTE function-of-function: float(sum(x))
   mean_x = sum_x/float(len(x_vector))                 # And again: float(len(x))
   sum_dev_sq = 0.0                       # place to keep sum of squared deviations
   for x in x_vector:                       # here's that step-through-a-list trick
      sum_dev_sq += (x - mean_x)**2                      # accumulate the devs**2
   variance = sum_dev_sq/float(len(x_vector))     # divide by the number of devs**2
   return variance                                           # deliver the answer!
```

Write a program called `vartest.py` that begins with this function definition. Then initialize a list of numbers, for example,

```
x = [1, 2, 2, 3]
```

and print its variance by writing

```
print("variance of x[] is %5.2f" % (var(x)))
```

The answer is 0.5. Make sure your version of the function returns that value, then try some different numbers in `x` (for example, [2, 3, 3, 4], or the negatives of those numbers, etc.). Note the wonderful fact that you can call `var()` with *any* list of numbers! It doesn't have to be named `x_vector`, which is a *private* name that `var()` uses *internally* to represent your list. It doesn't have to be any particular length. And even if it contains integers, the `float()` function will force every intermediate result to be represented as a floating-point number.

The "privacy" of what goes on inside functions is very important. None of the variables defined and used inside `var()` are "visible" to the outside world. After you have run `vartest.py`, you can see the values in `x` by typing "`x`" at the interactive Python Shell prompt. But if you type "`sum_x`" or "`variance`", then the Shell will complain that these variables are *not defined*! Note in particular that the `x` used in the `for` loop inside the *function* `var()` is a simple scalar numeric variable that holds (one at a time) the individual elements of `x_vector`, which is a copy of the *list* that your *program* calls `x`!

	White						
Red	1	2	3	4	5	6	sum
1	547	587	500	462	621	690	3407
2	609	655	497	535	651	684	3631
3	514	540	468	438	587	629	3176
4	462	507	414	413	509	611	2916
5	551	562	499	506	658	672	3448
6	563	598	519	487	609	646	3422
sum:	3246	3449	2897	2841	3635	3932	20000

3.5 LAB 3: Were Wolf's dice fair? An alternative approach using the variance.

As you know, Wolf tossed his pair of dice 20,000 times and obtained the results shown once again in the table below. If the dice were fair, then each face of each die should appear on about 1/6 of the 20,000 rolls, or 3333 times. There is no reason to expect Wolf's data to agree exactly with this expectation because, after all, dice are random. But several of Wolf's marginal counts do look suspiciously far from 3333. The question is, are these discrepancies larger than they ought to be if the dice were really fair?

In a previous lab (JEPy 2.5, page 14) you addressed this problem by tabulating counts. Here we will take a different approach. Consider what we might expect of an *unfair* die. With such a die, some faces appear frequently and others rarely. There should therefore be large differences between the counts for different faces. We can measure this effect by calculating the variance of the counts for the 6 faces of each die. For example, the counts for the white die are 3246, 3449, 2897, 2841, 3635, and 3932. The variance of these numbers is 150531.56. If Wolf's white die were fair, then we should expect to see comparable values fairly often in experiments with fair dice. But if Wolf's die were unfair, his variance would turn out to be unusually large.

To study this question, you will modify the Python program shown below. This code is available on the course web site, where it is called `Wolf_2_incomplete.py`. To download it, find it on the web site, right-click, and select "save as." Then run it to see what it does.

(Alternatively, you could copy the text below and paste it into the Python program editing window. The indenting and blank lines will probably disappear, but in that case you can just replace them by hand.)

```
from random import random                                    # line number  1

# define a function to return the variance of values in xvec                #  3
def var(xvec):                                                              #  4
    m = msq = 0.0                                                           #  5
    for x in xvec:                                                          #  6
        m += x                                                              #  7
        msq += x*x                                                          #  8
    n = float(len(xvec))                                                    #  9
    m /= n                                  # mean                          # 10
    msq /= n                                # mean square                   # 11
    return (msq - m*m)                      # variance                      # 12

w = [3246, 3449, 2897, 2841, 3635, 3932]    # counts from white die         # 14
r = [3407, 3631, 3176, 2916, 3448, 3422]    # counts from red die           # 15

Vw = var(w)                                                                 # 17
Vr = var(r)                                                                 # 18
print("Var(white):", Vw)                                                    # 19
print("Var(red)  :", Vr)                                                    # 20

nreps = 10                          # adjust this to do more replicates     # 22
for rep in range(nreps):            # outer loop: replicates of expt.       # 23
    count = [0,0,0,0,0,0]           # count[i] accumulates the numbers of   # 24
                                    #     rolls that showed i+1 spots       # 25
    for roll in range(20000):       # inner loop: rolls of the die          # 26
        spots = int(6.0*random())   # uniform on [0,1,2,3,4,5]  fn(fn())    # 27
        count[spots] += 1           # accumulate spot numbers               # 28

    v = var(count)                  # and here's our function call          # 30

    print("Replicate # %d: var=%f" % (rep, v))    # REMOVE ME later         # 32
```

When you run this program, you should get 12 lines of output. The first two report the variances of Wolf's two dice. The next 10 show the results of 10 simulated tosses of a fair die. The first line of this

program makes the "random" function (a member of the "random" module) available to our program. Lines 3-12 define a function that calculates the variance of a list of numbers. (Note that it's simpler in some ways than the one you implemented in section 3.4, pages 17-18, above.) Lines 14-15 define two Python lists, which contain the counts from both of Wolf's dice. Lines 17-18 calculate the variances of these lists, using the `var()` function defined in lines 3-12. Line 22 declares a variable called `nreps` which is short for "number of replicates" (repetitions of the experiment). This is the number of times we will repeat Wolf's entire experiment. Initially, `nreps` is only 10, because we don't want to drown in output. You will eventually want to make this number larger, in order to get accurate answers.

Lines 23-28 define a loop which is executed `nreps` times, once for each simulated repetition of Wolf's experiment. The loop is controlled by the first line, which uses `for` and `range`. Within the loop, line 24 defines and initializes a list called `count` which will keep track of the number of times each face of the die appears. Lines 26-28 do Wolf's experiment, and line 30 calculates the variance. Finally, line 32 prints a line of output. You will want to delete or "comment out" this line when you increase `nreps`.

To complete this lab, carry out the following steps and document your work.

1. Add a variable called `tail_r` (for "tail of the red-die distribution"). This variable should initially equal zero. Each time through the loop, if `v` is greater than or equal to `Vr`, then add 1 to `tail_r`. Thus `tail_r` will count the number of repetitions in which the simulated variance is at least as large as the observed one. Before you modify the code, think carefully. Where should `tail_r` be defined and initialized? Inside the inner loop? Before the inner loop but inside the outer one? Or before both loops?

2. Add the variable `tail_w` to keep track of how often the simulated variance is at least as large as `Vw`.

3. At the end of the program, and outside of the `for` loop, print two lines of output, one telling us what fraction of the time the simulated variance exceeded `Vr` and the other doing the same for `Vw`. The print statements should include text that explains what is being printed.

4. Once this works, take out the `REMOVE ME` line (or convert it to a comment), and increase `nreps` first to 100 (make sure that everything still works) and then to 1000. If you go much over 1000 then your program may take a long time to run.

5. Use your results to figure out whether either or both of Wolf's dice were unfair. Hand in your final program, its output, and a paragraph explaining your conclusions.

****

Chapter 4. Review: Eight big ideas, the Meaning of the Universe, and *where to put stuff!*

Congratulations! By now you've done a few introductory lab exercises and know something about the elements of the Python language. This chapter briefly reviews some of the central principles. Reading it may help you consolidate what you've learned so far mostly by example, and it may also help you prepare for what's to come.

4.1 Representing things with *values, variables* and *expressions.* Computers store information in sequences of binary "bits" (1 for "on" or 0 for "off"). Software then *interprets* these bit patterns as representations of the things

we're really interested in, which are most fundamentally *numbers* (like 42) and *strings* of text characters (like "spam"). The numbers are of two kinds: *integers* which can be represented exactly (the meaning of the universe is 00000000 00000000 00000000 00101010 in the standard 32-bit notation for integers), and *floats* which approximate real numbers (using a form of scientific notation that we won't bother to explain). In the representation of a *string*, each *character* is often encoded as single 8-bit "byte" (in this code the first letter of "spam" is 01110011, but if interpreted as a number it would be 115).

Python does the interpretation for us so we don't have to think about bits and bytes and such. But we do have to let Python know what kind of data "object" a given *variable* or other *expression* represents. A *variable* is just a *name* that Python associates with the *value* of a data object of a certain kind. If we type `x = 2`, then Python associates the name *x* with the *integer* value 2. But if we type `x = '2'` or `x = "2"`, then the value of *x* is the *string* 2, which is text, not a number of any kind.

Values are represented most simply by "literal" or "constant" values like `2`, `4.0e-001`, and `'spam'`, or they may be built up through the use of appropriate *operators* (as in `2 + 4.0e-001`) or *functions* (as in `sqrt(2)`, which has the value 1.41421...). If x = "sp" and y = "am", then x+y is an *expression* that *evaluates* to "spam".

4.2 Doing things to values with *statements*. Values exist but they accomplish nothing until we instruct Python to change or use them in some way. When we say `y=sqrt(x)`, we are using an *assignment statement* to create the variable `y` and associate the value `sqrt(x)` with it. If we say `print(2.0*y)`, we are again expressing a value (`2y`) and using a statement (`print()`) to make Python show it on the screen. Note: We do *not* need to know in advance what the value of `x` will be! Programming allows us to write general *algorithms* for transforming wide ranges of *inputs* into equally wide ranges of *outputs*, according to the *same* fully specified rule.

4.3 Organizing data structures with *lists*. Simple variables such as `x` allow us to use the same expressions and statements to refer to many possible values, as with our old friend `y=sqrt(x)`. Similarly, compound variables such as `x[i]` allow us to refer to *lists* of values. Such collections may become very large, but the algorithms needed to operate on them remain simple if each element (or group of elements) in the *data structure* can be transformed by the same procedure.

Python has several kinds of compound variables, but the most important by far is the *list,* which you have already met (in chapter 2 of *JEPy*). (Tuples and dictionaries are also important, and you should learn about them, but you won't need to master them for this course.) In Python, lists are one-dimensional arrays of variables that may refer to objects of any kind. Most often, the members of a list are numbers or strings. For example,

```
x = [12, 9, 0, 4.2, 3]
words = ['parrot', 'dead', 'spam', 'eggs', 'grail']
```

But they don't have to be all of the same kind!

```
answer = ['meaning', 42]
```

And most usefully, the members of a list can be *other lists!*

```
my_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

The elements of a list are referred to by indexes that begin at *zero*. What are the values of `my_matrix[0][0]` and my_`matrix[1][2]`? What is the value of `words[2]?` And what is the value of `words[2][3]?`

4.4 Iterating lists with *for* statements. We often want to step through the elements of a list. This could be done with a `while` loop that increments an index variable as in coin2.py on page 11 of *JEPy*. But it's much simpler and more elegant to say

```
for word in words:
     print(word)
```

For each of the five elements of `words`, sequentially, the `for` statement associates the new variable name `word` with that element. The `print` statement then causes that element to be printed to the screen.

The function `range(n)` creates the list of `n` consecutive integers beginning with 0. Thus `x = range(8)` assigns the list `[0,1,2,3,4,5,6,7]` to `x`. In the most popular looping construct, `for` iterates on a `range`:

```
for i in range(5):
     print(i, words[i])
```

The ***list comprehension*** (another feature unique to Python) is an implied `for` loop that creates and initializes a list. Thus `x = [i for i in range(8)]` is the same as `x = range(8)`, and

```
my_matrix = [[1 + j + 3*i for j in range(3)] for i in range(3)]
```

creates the list of lists that is written out in section 3, above. Try this, with modifications, if the syntax isn't clear.

4.5 Addressing the *pieces of a string* with *indexes* and *slices*. A string is a sequence of characters that can be referenced individually by subscripts, just like the elements of a list.

```
>>> word = 'spam'
>>> print(word[2])
a
```

But you can't directly change the individual elements of a string, so `word[3] = 't'` won't work. Instead, you can make a new string by extracting the first three characters with a *slice*, which can then be combined with the part to be changed. The expression for this value can then be assigned to the original variable name, if desired. Thus `word = word[0:3]+'t'` changes the value associated with `word` from `'spam'` to `'spat'`.

Slice notation is unique to Python, and it is extremely useful with lists as well as with strings. Any segment of any sequence-type expression can be referred to as `[<first position>:<last position + 1>]`. The value of `word[0:3]` is `'spa'`, and the value of `words[2:4]` is the list `['spam', 'eggs']`.

4.6 Modularizing the work with *functions*. These are self-contained blocks of Python-speak that do something and then deliver the result. Typically they take one or more input "arguments" and transform them into an output (usually a value, as in `y=sqrt(x)`, but sometimes a side effect such as printing something to the screen).

Some functions are built in to the core of Python (*e.g.*, `int(x)`, `float(x)`, `abs(x)`). Some functions must be loaded into memory from modules (*e.g.*, `random()` and `randint(i,j)` from the `random` module, or `sqrt(x)`, `log(x)` and `pow(x,y)` from the `math` module). And you can write your own, using the reserved keyword *def*, which may not be used for any other purpose. Here's the example from section 3.4, page 17, above:

```
def var(list_of_nums):
   sum_x = float(sum(list_of_nums))
   mean_x = sum_x/float(len(list_of_nums))
   sum_dev_squared = 0.0
```

```
    for x in list_of_nums:
        sum_dev_squared += (x - mean_x)**2
    variance = sum_dev_squared/float(len(list_of_nums))
    return(variance)
```

If `my_data` is the list [0, 1, 0, 1] (or those four numbers in any order), then `var(my_data)` evaluates to 0.25. This is very handy, because you can use `var` at any number of places in your program just by "calling" it:

```
my_variance = var(my_data)
```

4.7 Lists and strings do amazing things with their *methods*. These are much like functions, but they are tied more closely to properties of the object, and they sometimes *modify* the object instead of "returning" a new value derived from it. The methods available for *lists* are the most powerful and most frequently used. For example,

```
>>> words.sort()
>>> words
['dead', 'eggs', 'grail', 'parrot', 'spam']
```

The value of `words` is now changed to a list of the same five strings, but in alphabetical order (the default, but other options exist). *Lists of numbers* can also be sorted (in numerical order, of course, not alphabetical order).

```
>>> words.reverse()
>>> words
['spam', 'parrot', 'grail', 'eggs', 'dead']
```

Kind of obvious, I guess. Another useful list method is `append`, which adds its argument to the end of the list.

```
>>> words.append('Brian')
>>> words
['spam', 'parrot', 'grail', 'eggs', 'dead', 'Brian']
```

Some methods are more function-like, in that they return a value derived from the object *without* changing it. The `count` method provides a good example:

```
>>> words.count('eggs')
1
>>> words.count('cheese')
0
```

The `index` method can be extremely useful. It locates the first instance of a given element in the sequence:

```
>>> j = words.index('eggs')
>>> print(j)
3
```

Strings also have the `count` and `index` methods (and many others – we're just scratching the surface here).

```
>>> inspired = 'We believe these truths to be self-evident'
>>> inspired.count('e')
10
>>> inspired.count('e ')
4
```

Note that in the second case the argument is a two-character substring. The `index` method for *strings* also works as you might hope:

```
>>> inspired.index('e ')
1
```

But suppose we wanted to find the location of the *third* word ending in "e"? Fortunately, the `index` method will accept a second argument that tells it *where* in the sequence to start searching:

```
>>> j = 0
>>> for i in range(3):
    j = inspired.index('e ',j+1)
        print(j)
1
9
15
```

This algorithm searches three times, beginning initially at position 1 (the first possible location of a word-ending "e"). Then in the *subsequent* iterations, it begins one position *after* the *previous* word-ending "e". By this method it finds and reports the locations of the word-ending "e" in "We", "believe" and "these". Note that the algorithm would work on *any* text string with at least three words that end in "e" followed by a space.

4.8 Controlling the flow with *indenting*. You don't just want things to happen – you want them to happen in a particular order, and often you want them to happen only if some particular condition is true. In Python, the "flow of execution" is controlled in part by the order in which statements occur within a block of statements (from top to bottom, running down the page, as in all programming languages). But flow is also controlled very importantly by indentation, and Python is almost unique in this respect. You're already familiar with this feature of the language, but it always gives beginners fits, and it's a frequent cause of bugs even for experts.

Problems often arise when there are several levels of looping and/or conditional execution, in a hierarchical pattern. For example, suppose we wanted to calculate the mean value of the largest random number in a set of ten random numbers. Let's collect a million such "extreme values" and average them. This little program has been extensively commented to emphasize the hierarchically nested flow of execution (with comments that are themselves indented, to emphasize the point).

```
# extremo.py  12 September 2011
# Import the random-number generator *before* it's needed.
from random import random
# Initialize memory variables for the extreme values *before* the case-loop begins.
sum_ex = 0.0
n_ex = 0
# Begin the 1000000 cases.
for case in range(1000000):
    # Initialize memory for the largest random number just before *each* set of 10.
    max_x = 0.0
    # Now draw the 10 random numbers in a set.
    for i in range(10):
        x = random()
        # Test to see if x is the largest value yet (in *this* set of 10).
```

```
        if x > max_x:
            # Update max_x only when this is true.
            max_x = x
        # NOW WE RETURN TO x = random(), ABOVE, UNLESS i == 9!
    # We fall out of the i-loop when 10 random numbers have been drawn (i == 9).
    # Now max_x is the largest random() in a set of 10.
    # Use it to update the sum, and increment the sample size.
    sum_ex += max_x
    n_ex += 1
   # NOW WE RETURN TO max_x = 0.0, ABOVE, UNLESS case == 999,999 (= we've done 1M cases).
# We fall out of the case-loop when 1,000,000 sets of 10 have been drawn and processed.
# Note that the case-loop is the *outer* loop, which encloses the *inner* i-loop.
# Now we can calculate and print the mean, using the outer-loop memory variables.
mean_ex = sum_ex/float(n_ex)
print("Mean of %d largest random()s in a set of 10 is %7.5f" % (n_ex,mean_ex))
```

The answer I got was 0.90909. By thinking hard about the meaning of expectations, you may be able to explain this. Remember, `random()` returns a *uniformly* distributed number on the interval between 0 and 1, and in each set the algorithm collects 10 of them. *JEPr* may provide some help.

4.9 Where to keep your Python programs and the files they use or make. If you ask Python to import a module, or a function from a module, it will ***look first*** in the ***working folder*** where your program is located, and then in some other standard "system" locations, beginning with the Python installation itself. That's where it finds the random() function, for example, when you write "from random import random". You could put Alan's pgen.py module there too, but please don't. The installed software zone is best protected from accidental damage by remaining off the beaten track.

Instead, you should keep a copy of pgen.py in the ***working folder*** where your programs that use it are located. That working folder would most logically be placed on the Desktop, or else somewhere in your Documents folder. When you start to create a new Python program, you will use the editing window's **File > Save As** menu to navigate to your ***working folder*** and save your program there, under a unique new name. Then, when your program says something like "from pgen import bnldev", Python will find it right there in your ***working folder***, and you'll know that the copy of pgen in there is the one that your program is using. Unfortunately, when you launch Python "cold" its default behavior is to put your new program in the system folder where Python and its resources are installed. So to stay safe, keep all of your own programs, data files, and modules such as pgen.py in your working folder.

When a program uses the "open" command to prepare an existing file for input, or to create a new empty file for output, Python will attempt to do that in the folder where the program is running. To avoid confusion and accidents, make sure this is your ***working folder***, not the folder where Python is installed.