

Hacking Camp

a Teacher's Manual for charismatic hackers

10-hour course on serious programming for kids
with Python on the GNU/Linux system

Mark Galassi
Los Alamos National Laboratory

`mark@galassi.org`

August 13, 2018

Contents

Motivation and prolegomena	iii
1 The hardware and the operating system	1
1.1 Having students bring hardware	1
1.2 Start the installation	1
1.3 Opening up a computer to demonstrate its parts	3
1.4 Lecture on the blackboard	3
1.4.1 Hardware block diagram	3
1.4.2 Software functional diagram	4
1.5 After the installation	6
2 Start programming in Python	8
2.1 Concepts before we start	8
2.2 Learning to type and learning the editor	9
2.3 Reiterating “hello world” and starting loops	9
2.4 Introducing strings and lists	12
2.4.1 Strings	12
2.4.2 Lists	12
2.4.3 All sorts of types	13
2.5 Start talking about tic-tac-toe	14
3 Functions to do tasks, more tic-tac-toe	16
3.1 Structuring the program	18
4 Tic-tac-toe: playing moves	21
4.1 Taking input moves from the players	21
4.2 Improving flow and robustness	22

5 Tic-tac-toe: simplest computer play	26
5.1 First found	26
5.2 Random play	27
6 Checking if someone has won	30
7 Tic-tac-toe: more intelligent-play	33
7.1 Defensive play	33
7.2 Opportunistic play	35
7.3 Concluding words	35
A Life after the course	37
B Notes on installing GNU/Linux in class	38
B.1 Issues with old/cheap hardware	38
B.2 The Asus X551MA laptop	39
B.2.1 Preparing a USB memory stick	39
B.2.2 Saving off laptop info and restore drive	40
B.2.3 Booting from a USB stick	40
B.2.4 Installing GNU/Linux	40
B.3 The old Mac G4 PowerPC laptop	41
B.3.1 Preparing an installation CD	41
B.3.2 Booting from CD and installing	42
B.3.3 Post installation	42
C Software Freedom	44
D GNU Free Documentation License	45
1. APPLICABILITY AND DEFINITIONS	45
2. VERBATIM COPYING	46
3. COPYING IN QUANTITY	46
4. MODIFICATIONS	46
5. COMBINING DOCUMENTS	47
6. COLLECTIONS OF DOCUMENTS	47
7. AGGREGATION WITH INDEPENDENT WORKS	48
8. TRANSLATION	48
9. TERMINATION	48
10. FUTURE REVISIONS OF THIS LICENSE	48
ADDENDUM: How to use this License for your documents	48

Motivation and prolegomena

The joy and beauty of serious programming

Computer programming, like music, chess and mathematics, can be taught to young children. These children can often perform at the same level as adults in many areas of programming.

Many courses are taught to kids. The ones I have seen are very well developed and organized. They use a friendly programming environment in which users see a rapid visual response to their commands and snippets of code.

Unfortunately these environments do not bridge a crucial gap to programming the way it is done by professionals. The language is not one used for serious programming, the environment does not scale to more complex software development, and the graphical output, while immediate, does not then translate into production-level graphical user interfaces. When the students are confronted with real technical work, for example in a college or graduate school internship, they can feel a sense of loss that those tools did not prepare them for their “real work”.

Important tools for a programmer include at least the following¹:

- a deep understanding of how her computer and operating system work
- mastery of one or more general purpose programming languages
- a strong connection to her development tools (most importantly the programming editor)
- comfort with how the command line brings together a variety of small tools

I have carried out various experiments in teaching this more advanced approach to students, both in the 8- to 12-year-old range and to teenagers.

¹Of course this list could be arbitrarily long; I will focus on these items for now.

Some of the approaches I have tried have not worked too well, but the experiments led me to an approach which has been quite successful: the students learned the material readily and surprised me and others with how quickly they grasped real programming tools.

An important matter comes up with teaching real-world programming, and this is not a problem unique to kids. It is the problem of understanding how computers work. When I learned to write code in the early 1980s everyone who had a computer at home had to understand how it worked and had to put together many components.

Today both children and adults can use a laptop or tablet computer to good purpose for several tasks (video conferences, reading mail, accessing web resources, . . .) without understanding how the computer works or even *what the parts of the hardware and software are*.

The analogy I have come up to explain this is that of car ownership. A person can drive a car with an automatic gear shift and have no idea: (a) that there are gears with a different ratio between how fast the engine turns and how fast the wheels turn, (b) how to fix a car if it develops problems, (c) how to design and build a car.

Analogously to meeting kids who don't know who The Beatles are², today's computer users might not know the meaning of CPU, RAM, graphics cards, network controllers, hard disks. Moving up from hardware they might not know where the operating system fits into a functional diagram of computer hardware and software, or what an editor is. They might not even know what a "computer program" is.

These are all important things to understand when you write code, so we will have to start our training by understanding how hardware and software work under the fancy cover.

This is why the course requires students to find an old computer or laptop: in the first lesson the students will open up a computer to see what the parts are inside (with the instructor's guidance), and they will then install a GNU/Linux distribution on their own computer so that they can do their programming on a machine they have configured, starting from the bare hardware.

And now for the beauty and joy. Serious programming is not only fun: you can experience the deep joy of creating something or solving a problem. We hope to do some of that in the course, so we should emphasize to the students that this activity has deep rewards.

²A rock and roll band from the 1960s and early 1970s.

Automation of repetitive tasks

People who have programming as a tool in their belt know that the main function of computer programming is the automation of repetitive tasks.

This is so important that there is a mantra which I repeat very often when I teach: “the reason we use computers is for the automation of repetitive tasks”. When we introduce loops, and at many other times, I have the class repeat “automation of repetitive tasks”. It eventually becomes a humorous mantra, *with the joke being on the instructor* because she repeats herself so many times, but the phrase will be remembered.

Here is how to test whether students have understood this. After they have started using the shell, have them create a bunch of files with a typo:

```
for i in 0 1 2 3 4 5 6 7 8 9
do
    fname=file_number_$i
    echo "the number is $i" > $fname
done
```

Then tell them “now you realize that you misspelled and wrote *file* instead of *file*. What will you do about it?”

At this point they will not have the skills to write a script to fix all the file names, but they can still give this correct answer: “the right way to fix it is to write a script of some sort.”

At the end of the class you might want to give give them the script that fixes all the file names. There are many ways to do it — I usually do this:

```
for bad_fname in fille_number_*
do
    good_fname=`echo $bad_fname | sed 's/fille_/file_/'`
    echo "renaming file from $bad_fname to $good_fname"
    mv -i $bad_fname $good_fname
done
```

Plan

This book has five chapters which correspond to five two-hour lessons. I have taught this course in two formats: as a week-long class in which kids show up for two hours/day (appropriate for a break from school), and as a weekend-intensive course in which you have to pack ten hours into two days.

It is important to follow the plan and keep a momentum in the class, completing each lesson on time. This can be difficult in case some students

need a bit more time, but we are aided by a very fortunate property of people with technical passion: they love to share their knowledge and help others. At the end of a class the students can help others catch up.

Since serious computing work involves solving real open-ended problems, rather than “canned” exercises, the instructor should be available in between classes to make sure that everyone can catch up and be ready for the next class without feeling lost.

But there is a flip side: this course is aimed at students who are self-motivated and ready to step out of their comfort zones. Some students might not be ready for this. Because of this I make it clear, when I advertise the course, that students will have to work hard and step out of their comfort zone in learning new ways of computing.

Why use GNU/Linux and Python?

The motivation behind learning the GNU/Linux operating system is that:

- the GNU/Linux operating system is the basis for almost all the computers that “run the world”: from the servers at Amazon and Google to the supercomputers that power science, the operating system is almost always GNU/Linux
- the GNU/Linux operating system is written by programmers for programmers: it offers a choice of several delightful programming environments and puts few barriers in the way of the programmer
- the GNU/Linux system is free (as in freedom), but it is also free as in cost: the entire system and its wealth of tools (for programming and all else) are available free of cost
- the GNU/Linux system has been packaged in many ways, some of which work quite well on very old hardware with less memory

The motivation behind teaching Python is:

- it is a pleasant and easy language to learn
- it is also a language used for many industrial-strength applications
- it can be used for a wide variety of interesting programs

We will be using version 3 of Python (python3) which is now becoming quite widespread. It has some incompatibilities with python2, but they are truly minor (we will only notice it in the `print()` function) and it is good to start using the new way of doing those things.

Preparation for class: venue and materials

If you get a group of kids signed up for the course you need to be ready to make the class time move at a good pace. If time is lost getting equipment to work or figuring out the network then you will lose the kids' attention.

I do a careful preparation of the site: I verify that we have electrical outlets, that we have hard ethernet as well as wi-fi, that there is a blackboard or whiteboard, tables, chairs, . . .

I also correspond extensively with the kids (or their parents for the younger kids), making sure that they have obtained a piece of hardware and finding out what kind of computer it is.

Then I have a couple of boxes of equipment I bring to the classes with (at least) the following items:

- an ethernet hub with as many ports as there are students
- dry erase markers
- several USB mice
- loaner laptops: a combination of cheap ones I purchased with a grant for the course, and of old computers the center had sitting around
- CDs and DVDs with several GNU/Linux distributions (labeled)
- USB flash drives (memory sticks) with several GNU/Linux distributions (labeled)

I also do a test install of the operating system on at least one of each model of loaner laptop, and I write down the procedure.

Using, adapting and improving this book

I have written this teacher's manual in case you want to use it as a starting point to develop our own course for children. You may obtain a copy of the source document for this manual³ and modify it to fit your needs, and

³<https://bitbucket.org/markgalassi/hackingcamp-teacher-manual>

even redistribute your modifications (see Appendix C for the license on this document).

I have not written this book to be a tutorial for children. You, the reader, are going to be tutoring the children, so this book is written to give you an outline and many possible details on what to do.

The target audience (that's still you) is someone who has at least some small experience programming. In that case the examples in this book will make sense to you and you will be able to explain them to the kids, even as you are learning yourself!

There are some areas you will most likely need to adapt, such as Appendix B which I tailored to the specific hardware we had available in the classes I taught. Other areas you might want to adapt to fit your style or your audience.

And naturally if you have ideas for improving the course or developing follow-on courses, please be in touch with me!

Lesson 1

The hardware and the operating system

1.1 Having students bring hardware

An essential part of this course is that the students should install an operating system themselves onto bare metal. This will give them a much better overall view of what the hardware + software combination that is a computer is made of.

There are several side benefits: *(a)* the students are exposed to a free software environment, *(b)* people who teach Python classes always grapple with installing and using Python on widely different systems — we will not have this problem, *(c)* the students become versed in the UNIX and GNU/Linux way of computing and learn some system administration skills.

In announcing the course I urge students to bring in an old computer, such as “grandma’s or grandpa’s old laptop”. The idea is to encourage them to ask around: many families will have old computers sitting around, and if not their neighbors probably do. But if they do not it is still worth having students take such initiative. Teachers can also raise grant money to find laptops for students, and I have found several very usable computers with the *freecycle* mailing list.

1.2 Start the installation

Installation instructions for the computers I have used in my course are in Appendix B. I have also had very good luck getting members of the local GNU/Linux user group in Santa Fe to come help with the installation.

It is important to get to this point rather early in the first 2-hour lesson, since this is rich ground for unpredicted problems. The students might bring hardware that does not work, or that is so strange that it is hard to install GNU/Linux on it (although that is rare these days). Or they might have brought something that is more than 10 years old, for which you would need a truly tiny GNU/Linux distribution.

To limit the delays that can come from this I recommend having some “loaner” laptops available for students to use in class if it turns out that their computer is just not working. If the installation appears to not be working then make sure you shift to using your loaner laptops.

As for a choice of distributions: I show up with several CDs/DVDs and USB memory sticks so that I am ready to install many different styles of GNU/Linux.

I also have a couple of USB wi-fi adaptors that are known to work without special drivers, since certain laptops have built-in wi-fi that requires special drivers.

Which distribution to install depends on how much RAM is in the laptop. I have typically installed the following distributions:

less than 2 GB RAM xubuntu 14.04

2 GB and 3 GB RAM xubuntu 16.04

4 GB and more ubuntu 16.04

As you can see, I have mostly gravitated toward the ubuntu and xubuntu distributions. This is not because I have a preference for these distributions or desktops. The reason I choose them is because they (a) Ubuntu installers do a reasonable job of detecting laptop hardware, (b) they are solidly maintained, with 14.04, 16.04 and 18.04 being *long term support (LTS)* distributions which will be maintained for five years, (c) they are widely used the students can get good support, and (d) there is a vast choice of desktop environments to run.

In class we will mostly use a programming editor and the Python interpreter, so the style of desktop doesn’t really matter that much.

The installation can take from a few minutes to half an hour, and we will use this time to go to the front of the class and discuss computer hardware.

1.3 Opening up a computer to demonstrate its parts

This “show and tell” should be done with a desktop computer, preferably a pretty old one, since the parts are more spread out and easier to identify.

The computer should be opened on a desk near the blackboard so that you can draw a hardware block diagram (see Section 1.4.1) at the same time.

You should identify the CPU, RAM and peripherals, discussing how they are connected to the CPU. Sometimes a peripheral is on the same board, and hence is connected via short solder connections. Other times it’s on a separate card, plugged in to the “bus” (in most modern PCs it’s a PCI bus).

Here is a basic list of some of the components you might identify:

backplane/bus where the cards are seated

motherboard the board with the CPU, much of the RAM, . . .

CPU the biggest item on the motherboard

RAM the computer’s fastest memory, sits on the motherboard in DIMMs — mini-cards, short and wide which plug in to the motherboard

graphics find the graphics chipset, either on the motherboard or in a separate card (it might be useful to bring two different computers)

hard disk

optical drive CD/DVD

network port

serial port

video connector

1.4 Lecture on the blackboard

1.4.1 Hardware block diagram

While you have the “show and tell” computer open (Section 1.3) you should draw a block diagram of how the hardware fits together. Many examples of block diagrams can be found on the web, but they often go into too much detail. I use a simple block diagram, a bit like the one in Figure 1.1.

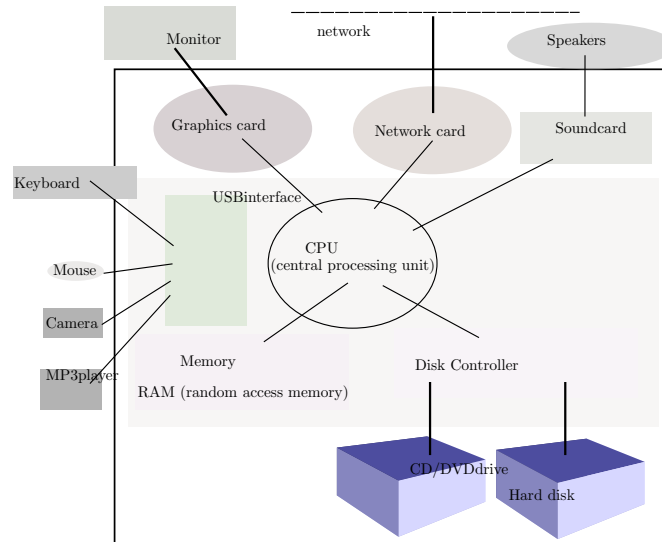


Figure 1.1: A diagram of the hardware parts of a typical personal computer.

Once you have the hardware block diagram on the blackboard you can animatedly talk about how each of those components is related to what the children have seen in their daily life.

For example: “has anyone here ever seen a movie streaming on the network, like with Netflix?” and when someone says “yes” you can point to the portions of the diagram which shows the ethernet port and the graphics card and the cable that connects to the monitor.

Or: “has anyone here ever looked at a weather forecast?”, at which point you can point to the CPU and RAM and talk about how those forecasts come from a lot of very intense calculations inside the CPU.

The goal of this kind of interactive dialogue is to get the students to feel that they part of the class, and also to make their subconscious sense a connection between this understanding of the hardware and their every day experience.

1.4.2 Software functional diagram

Draw a diagram similar to that in Figure 1.2.

In explaining this to young people, describe the layers from the inside out. The operating system sits between the hardware and the application programs and provides a uniform way for programs to access hardware resources.

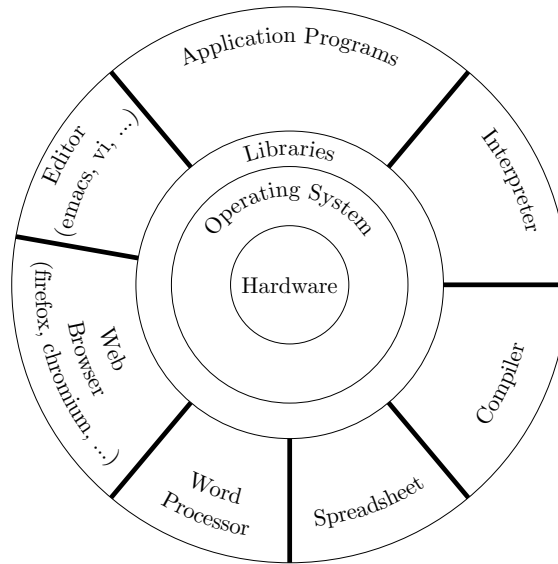


Figure 1.2: A diagram of the layers of software above the hardware.

To make this part interactive you can ask “who can tell me what an example of an operating system is?” Kids will answer “Windows” or “Macintosh”. I find it useful to point out the importance of free software operating systems by saying something like “very good”, and then writing down the list on the blackboard starting with the GNU/Linux operating system, and putting their responses below with the company that sells those operating systems. For example:

- GNU/Linux — used by programmers and on servers everywhere
- Microsoft Windows — used in many office and home computers
- Apple OS X — used in many Macintosh computers

and then I give a brief discussion (more will come throughout the course) of how the GNU/Linux system is designed and developed by volunteers with the goal of offering users a free¹ operating system that is not steered by a company’s marketing agenda.

Libraries allow higher level ideas to be expressed using basic building blocks. You can give the example of a task in daily life that involves numerous tiny steps, such as getting milk from the refrigerator. You could ask

¹“free” as in “freedom”; see Appendix C.

someone to get you milk from the refrigerator by saying: “please step forward 2 paces, then turn left 40 degrees, then step forward two more paces, then put your arm out, then close your fingers around the handle, then pull, ...”

Or you could simply ask someone “please get me milk from the refrigerator”. A library gives you higher level ways of expressing the task, delegating the small details to the library.

When you get to the “Application Software” layer remember that many of them don’t know what a “program” is. Or better: they probably know what a program is, but they don’t know it’s called a program. You can give them examples of programs they have seen in school or they might have used at home.

1.5 After the installation

Having finished our lecture on hardware and software architecture, we start telling the students to bring up a terminal.

Typing at the terminal and at a programming editor will be what we do all the time, so they need to start getting used to typing commands at the shell prompt in the terminal. Eventually they will develop a feeling for what the terminal and the shell are.

The first thing we can do is have them type:

```
$ tuxtype
```

The computer will tell them to install it with:

```
$ sudo apt-get install tuxtype
```

and they should do that. This is an entertaining “typing tutor” which will get them in the habit of touch-typing.

I let the students stay in `tuxtype` until about 5 minutes before the end of this first 2-hour lesson.

Now they have just a few minutes left. Remembering that everyone is relieved and more enthusiastic about a course when classes do not run over time, we can use these last minutes to:

- Have them bring up a terminal.
- Have them run “`sudo apt-get update`” and “`sudo apt-get install python3`”

- Have them bring up the Python interpreter by typing “python3” on the command line²
- Have them type their first line of code: “[language=python]print('hello world')”

This way the kids go home from their first lesson having run a brief program.

²We will use python3 in this course.

Lesson 2

Start programming in Python

2.1 Concepts before we start

There are (at least) two ways of having a computer run your Python program: you can type instructions in the Python interpreter, which only works for a few lines of code, or you can use a text editor (preferably a programming editor) to write the program, save it to a file on disk, and then run the Python interpreter on that file.

When I teach this I alternate between the two approaches: I might give a quickie to explore a feature of the language, in which case I will have students type it into the interpreter. But as the program grows I have them type it in an editor and execute it from the command line.

This means that the students need to get familiar with a few ideas and techniques. I start by explaining the following concepts at the blackboard:

- file
- shell
- how you can (and should) use the shell to do things instead of a graphical file manager
- editor

in addition to other ideas:

- programming language
- the literal-mindedness of computers

2.2 Learning to type and learning the editor

Students might not know how to type yet. I have them install the typing tutor `tuxtype` with “`sudo apt-get install tuxtype`”, after which they can run `tuxtype` on the command line to practice typing. I don't have them spend more than a couple of minutes on it at this time in class, but I tell slow typists that they need to practice.

I then have students install the programming editor `emacs` with “`sudo apt-get install emacs`” and spend a longer amount of time learning `emacs`. This can be done with the `emacs` tutorial. Initially students will try to rely on the mouse and the arrow keys, and a proficiency test would be if by the end of the week they are using native `emacs` navigation and other approaches from the tutorial.

Once they are comfortable creating a new file in `emacs` (`C-x C-f`) and saving it to disk (`C-x C-s`) we are ready to move on to writing code.

2.3 Reiterating “hello world” and starting loops

I write all the codes here on the blackboard during the lecture. I have the students input them, either at the Python interpreter or in an editor, to then be run from the command line.

At the command line type “`$ python3`” and at the python prompt type:

```
>>> print('hello world')
```

and then a few more instructions to see how to use Python as a simple calculator:

```
>>> 7*4
>>> print(7*4)
>>> 125/13.5
>>> import math
>>> math.sqrt(1.7 + 32/17.1)
```

Then introduce variables with:

```
>>> x = 7
>>> y = 4
>>> x*y
>>> print(x*y)
```

Then move on to something more interesting with:

```
>>> for i in range(10):
...     print(i, i*i)
```

This example allows the telling of humorous anecdotes about sadistic teachers. I mention that in the old days (including my own childhood) students were often given punishment for being unruly in class. Sometimes they were unruly because they were too smart for the course material, other times because they were just ill-behaved. A common form of punishment was to have the child write something repeatedly on the blackboard. My teacher would make me write 50 times:

I will not disturb in class

This brings us to the anecdote of Carl Friedrich Gauss (here I'll ask the kids "do you have a favorite mathematician superhero?" and run with that idea for a while...). The legend (possibly apocryphal – who knows what apocryphal means? what's a famous apocryphal story?) tells that his math teacher asked him (and maybe the whole class) to add all the numbers from 1 to 100 so that the teacher could take a break.

The teacher's break was brief: Gauss answered right away. What is the answer? If the class does not see it we can demonstrate it: write out $1+100 + 2+99 + 3+98$ and so forth to $49+52$ and $50+51$. That makes for $50*100$, which is 5050. More generally, Gauss got:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

So if you are given a repetitive task by a sadistic teacher, how do you handle it? Two solutions: either you come up with a cool math formula that lets you calculate it immediately, or you write a computer program to do it. That's what we did with our loop.

Now take that looping program and put it in a file called `loop.py`. The code in the file will look like:

```
for i in range(10):  
    print(i, i*i)
```

Listing 2.1: `loop.py` - first program

and you can run it with

```
$ python3 loop.py
```

Then expand on the program:

```
import math  
for i in range(10):
```

```
print(i, i*i, math.sqrt(i))
```

Listing 2.2: loop.py - adding code

Students might enjoy changing the 10 to be 100 or even 1000 to see what happens; some will plug in ridiculous numbers, which gives the teacher the opportunity to have the students run their CPU hard on a tight loop, put their ear close to the computer, and listen closely to the whining of the fan. Then hit `control-C` and hear the whine stop.

Now show the formula for Fahrenheit and Celcius conversion:

$$T_{\text{degC}} = (T_{\text{degF}} - 32) * \frac{5.0}{9.0} \quad (2.1)$$

and write the following program and save it to a file called `fahr.py`:

```
for degF in range(100):
    degC = (degF - 32) * 5.0 / 9.0
    print(degF, degC)
```

Listing 2.3: fahr.py

And of course here I point out that “sure, you could take a calculator and write down all the possible fahrenheit–celcius conversions, but ... [drum roll] *the purpose of computers is to automate repetitive tasks!*”

Now let us actually do Gauss’s task the hard way and the easy way. Put the following into `gauss-sum.py`:

```
sum_from_adding = 0
for i in range(1,101):
    sum_from_adding = sum_from_adding + i
print('hard way: the sum from 1 to 100 was: ', sum_from_adding)

sum_from_gauss = (100 * (100 + 1)) / 2
print('easy way: the sum from 1 to 100 was: ', sum_from_gauss)

if sum_from_adding == sum_from_gauss:
    print('yay! Gauss was right!')
else:
    print('hmmmf! Gauss was wrong!')
```

Listing 2.4: gauss-sum.py

2.4 Introducing strings and lists

2.4.1 Strings

I write on the board (and have the students type at the Python interpreter) several tutorial snippets to get students comfortable with strings.

```
>>> s = 'hello'
>>> t = 'world'
>>> print(s, t)
>>> s + t
## Oh cool: I can stop typing print() every time when I'm
## typing at the interpreter; from now on we will use print()
## in programs, but seldom in examples at the >>> interpreter
## prompt
>>> s + ' ' + t
```

2.4.2 Lists

First I do a “show and tell” with a shopping bag with three different types of things in it. They could be a pencil, a box of crackers, and a notepad. I then introduce the list as analogous to the shopping bag: it contains several elements of different types. (For now I gloss over the fact that a list is ordered while the shopping bag is typically not ordered.)

I then write on the board (and have the students type at the Python interpreter) several tutorial snippets to get students comfortable with lists.

```
>>> mylist = [2.5, 17, 'dude']
>>> print(mylist)
>>> mylist
>>> mylist[0]
>>> mylist[1]
>>> mylist[2]
```

AAAARGHH: repetitive task alert!!

```
>>> for i in range(3):
...     print(i, mylist[i])
>>> for item in mylist:
...     print('item is:', item)
>>> print(len(mylist))
>>> for i in range(len(mylist)):
...     print(i, mylist[i])
```

2.4.3 All sorts of types

We have seen a few data types already: integer, floating point number, string, list. Let's type some commands at the Python interpreter to get a better feel for these.

Now I write the following on the blackboard and the students type them into their python3 interpreter:

```
>>> type(4)
>>> n = 42
>>> type(n)
>>> type(4.4)
>>> x = 3.141592654
>>> type(x)
>>> type(2.0), type(2)
>>> type('hello world')
>>> s = 'hello world'
>>> type(s)
>>> mylist = [2.5, 17, 'dude']
>>> mylist
>>> type(mylist)
>>> mylist[0]
>>> type(mylist[0])
>>> len(mylist)
>>> type(len(mylist))
>>> mylist
>>> for i in range(len(mylist)):
...     print('index:', i, 'list-entry:', mylist[i], 'type:', \
            type(mylist[i]))
```

I then go to another portion of the board and ask “so what different types of data do we have in Python?” The audience might answer something like “numbers, strings, lists”. I would then say that it's a good start, but to note that there are two different types of numbers: floating point and integer. This comes up in particular when we print the type of 2 and of 2.0.

For younger kids I would digress briefly to mention that floating point numbers are the ones with a decimal point, while integers are “whole numbers”. It is always an interesting challenge to match terminology between programming languages and the particular terms a young student's math curriculum uses.

Then we can also discuss how to turn strings into integers. We will first introduce *conditionals* (if-statements), logic, and booleans (we don't need to call them booleans for younger kids).

```
>>> if 2 > 3:
...     print('the impossible just happened')
```

```
... else:
...     print('phew: 2 is not greater than 3')
>>> x = 7
>>> y = 8
>>> if x*y < (x+1)*(y+1):
...     print('that made sense')
>>> x, y
>>> x == y
>>> x, y
>>> x = y
>>> x, y
```

This gives us a chance to talk about the meaning of = (assignment) and == (test for equality).

Now we can explore types and type conversions:

```
>>> s = 'hello world'
>>> ns = '42'
>>> n = 42
>>> n, ns
>>> n == ns
>>> n, str(n)
>>> str(n) == ns
>>> ns, int(ns)
>>> n == int(ns)
```

These last sequences (and in fact all “type at the interpreter” sequences) should be accompanied by a lot of discussion:

1. You write the expression on the board.
2. You wait until they have all typed it in the interpreter.
3. You work slowly and carefully with a student who has not spoken up much recently to discuss what it means.

2.5 Start talking about tic-tac-toe

Our goal is to write a non-trivial program. We will write a program to play tic-tac-toe. Lessons 3, 4 and 7 will develop the program in detail.

But at the end of this lesson we should start whetting the student’s appetite by showing how we can use lists and strings to represent a tic-tac-toe board.

First ask “if you need to represent a row on a tic-tac-toe board, what would you use?” Then mention that it’s a sequence of three characters:

either an 'x' or an 'o' or a ' ' (space), and finally get to the choice of a list. Then we need three rows, so we will use a list of lists.

Let us turn that into code. Type these lines into the interpreter:

```
>>> row0 = [' ', 'x', 'o']
>>> row1 = [' ', 'o', ' ']
>>> row2 = ['x', ' ', 'x']
>>> board = [row0, row1, row2]
>>> print(board)
>>> for row in board:
...     print(row)
>>> for row in board:
...     for cell in row:
...         print(cell + ' ', end="")
...     print
```

Then open a file called `board.py` in emacs and type in this program:

```
row0 = [' ', 'x', 'o']
row1 = [' ', 'o', ' ']
row2 = ['x', ' ', 'x']
board = [row0, row1, row2]
print('-----')
for row in board:
    for cell in row:
        print('|', end="")
        print(cell, end="")
    print('|')
print('-----')
```

Listing 2.5: `board.py` - first stab

then save the file. In the terminal window run the command:

```
$ python3 board.py
```

which should give the following ascii output:

```
-----
| |x|o|
-----
| |o| |
-----
|x| |x|
-----
```

With this we stop for the day.

Lesson 3

Functions to do tasks, more tic-tac-toe

We have written some Python instructions to print out a tic-tac-toe board. There were about seven lines of code.

Suppose we want to print the board out often. Should we put in those seven lines of code each time?

The answer is “certainly not” (repeat the mantra on repetitive tasks). The programming language feature that helps us avoid this duplication is called a “function” (also called “procedure”, “subroutine” or “method”).

Let us start with simple mathematical functions:

```
>>> def f(x):  
...     result = x*x  
...     return result
```

the function takes a variable `x`, does things to it (in this case squaring it), and returns another value (the square of `x`).

Functions don't always have to be mathematical, nor do they have to return a value: they could make things happen on your screen. Here is how we can print a tic-tac-toe board. We will modify `board.py`:

```
def print_board(board):  
    print('-----')  
    for row in board:  
        for cell in row:  
            print('|', sep="", end="")  
            print(cell, sep="", end="")  
        print('|')  
    print('-----')
```

```

row0 = [' ', 'x', 'o']
row1 = [' ', 'o', ' ']
row2 = ['x', ' ', 'x']
board = [row0, row1, row2]
print_board(board)

```

Listing 3.1: board.py - with print_board()

We can now look at how to modify the tic-tac-toe board, and then see how functions help us with that.

Let us say that now it is o's turn to play and she places an 'o' on the middle square of row two. That would be row 2, column 1. The way we would do that in Python is by adding this at the end of the previous code snippet:

```

# ...
board[2][1] = 'o'
print_board(board)

```

Listing 3.2: board.py - setting a cell

and the “print_board(board)” will show us the new state of the board, which should now look like:

```

-----
| |x|o|
-----
| |o| |
-----
|x|o|x|
-----

```

We now write a function to set a position on the board. The entire program now looks like:

```

#!/usr/bin/env python3
def print_board(board):
    print('-----')
    for row in board:
        for cell in row:
            print('|', sep="", end="")
            print(cell, sep="", end="")
        print('|')
    print('-----')

def set_cell(board, row, col, val):
    board[row][col] = val

```

```

row0 = [' ', 'x', 'o']
row1 = [' ', 'o', ' ']
row2 = ['x', ' ', 'x']
board = [row0, row1, row2]
print(before:')
print_board(board)
set_cell(board, 2, 1, 'o')
print(after:')
print_board(board)

```

Listing 3.3: board.py - with print_board and set_cell

The main difference here is that instead of setting the cell with

```
board[2][1] = 'o'
```

we use

```
set_cell(board, 2, 1, 'o')
```

This does not look like a big savings, but this way of expressing things will have advantages later.

We are now ready to start playing out moves.

3.1 Structuring the program

With the code we have written in this lesson, starting from Listing 3.3, we can write a program which plays a sequence of moves and shows the board after each one.

First a discussion on the overall structure of a program. It's a good idea to define a “main function” in which you put the main flow of the program. This main flow should consist of a sequence of Python instructions, most of which should be function calls.

The following code shows a small example of this structure:

```

#!/usr/bin/env python3
def main():
    print('f(2.2) is ', f(2.2))
    print('f(2.7) is ', f(2.7))
    for x in range(5):
        print('f(', x, ') is ', f(x))

def f(x):
    return x*x

main()

```

It's a good time to discuss the invocation of `main()` at the end. Point out that the first two functions were *defined* but never *called*. At the end, after we define all our functions, we call `main()` to start the program.

Now let's move on to a structured program that uses `set_cell()` and `print_board()` to play out a sequence of moves.

```
#!/usr/bin/env python3
def main():
    board = new_board()
    print_board(board)
    # make a move as 'x'
    set_cell(board, 1, 1, 'x')
    print_board(board)
    # make a move as 'o'
    set_cell(board, 0, 1, 'o')
    print_board(board)
    # make a move as 'x'
    set_cell(board, 2, 2, 'x')
    print_board(board)
    # make a move as 'o'
    set_cell(board, 0, 0, 'o')
    print_board(board)

def new_board():
    """Makes a board where all markers are spaces"""
    row0 = [' ', ' ', ' ', ' ']
    row1 = [' ', ' ', ' ', ' ']
    row2 = [' ', ' ', ' ', ' ']
    board = [row0, row1, row2]
    return board

def print_board(board):
    """prints the current state of the board"""
    print('-----')
    for row in board:
        for cell in row:
            print('|', sep="", end="")
            print(cell, sep="", end="")
        print('|')
    print('-----')

def set_cell(board, row, col val):
    board[row][col] = val

main()
```

Listing 3.4: board.py - play a sequence of moves

One interesting thing to point out about the program in Listing 3.4 is that the main function can be read (almost) as if it were English, and this is one of the fantastic things about computer programming: if we structure our program well, then the program flow is expressed with great clarity.

This ends our lesson, which is one of the easier ones. Things will get more intense in the final lessons.

Lesson 4

Tic-tac-toe: playing moves

4.1 Taking input moves from the players

Now that we have seen how to program in moves I will tell the students: “you should be saying ‘but *Maaaark*, this is just a sequence of pre-programmed moves; nobody is playing!’ ” I then urge them to have patience because we will now start taking input from players.

First an example of taking input. This can be typed at the interpreter:

```
>>> row_str = input(row? ')
## [it will ask you for input; you should type a number 0, 1 or 2
## students might be confused and hit <enter> again without typing
## a number, so it bears some explanation and some retries]
>>> print('the row was:', row_str)
```

Remember that in Section 2.4.3 we discussed that strings (with digits in them) are different types from numbers. You can see that `row_str` is a string and not a number.

You can convert a string to a number with “`int(row_str)`”:

```
>>> row_str = input('row? ')
## [it will ask you for input; you should type a number 0, 1 or 2]
>>> row = int(row_str)
>>> print('the row was:', row)
>>> print('types:', type(row_str), type(row))
```

Let us now write a function which accepts a player’s move. We should create a new program file called `ttt.py`, starting from `board.py`. We can do this at the shell with:

```
$ cp board.py ttt.py
```

In emacs we can edit `ttt.py` instead of `board.py` in two ways: with `C-x C-f` to load the new file, or we can exit emacs (`C-x C-c`) and then (from the shell) re-run emacs on the new file:

```
$ emacs ttt.py
```

Now we add, between the functions `print_board()` and `set_cell()`, the following:

```
# [...]
def get_move(board, marker):
    """asks the player for a move and sets the appropriate cell"""
    row = int(input('row? '))
    col = int(input('col? '))
    set_cell(board, row, col, marker)
# [...]
```

This works and we can now rewrite the main program like this:

```
# [...]
def main():
    board = new_board()
    print_board(board)
    # player 1 move
    get_move(board, 'x')
    print_board(board)
    # player 2 move
    get_move(board, 'o')
    print_board(board)
    # player 1 move
    get_move(board, 'x')
    print_board(board)
    # player 2 move
    get_move(board, 'o')
    print_board(board)
# [...]
```

Listing 4.1: `ttt.py` with player input

The game now could go as shown in Figure 4.1.

4.2 Improving flow and robustness

The function we have written works well and our program now is equivalent to a piece of paper and a pen: it allows two players to input their tic-tac-toe moves.

But there are some problems with the `get_move()` function:

```

$ python3 ttt.py          | |x| |
-----                  -----
| | | |                  |o| | |
-----                  -----
| | | |                  please enter row: 0
-----                  please enter col: 0
| | | |                  -----
-----                  |x| | |
please enter row: 1      -----
please enter col: 1     | |x| |
-----                  -----
| | | |                  |o| | |
-----                  -----
| |x| |                  please enter row: 2
-----                  please enter col: 2
| | | |                  -----
-----                  |x| | |
please enter row: 2     -----
please enter col: 0     | |x| |
-----                  -----
| | | |                  |o| |o|
-----                  -----

```

Figure 4.1: A run of the tic-tac-toe program with two players alternating moves.

- It does not check that the row and column are correct (i.e. equal to 0, 1 or 2).
- It does not check if the cell was already occupied! This allows cheating.

A version of this function which checks on the row and column is:

```

# [...]
def get_move(board, marker):
    """asks the player for a move and sets the appropriate cell"""
    row, col = -1, -1
    while not row in (0,1,2):
        row = int(input('please enter row: '))
    while not col in (0,1,2):
        col = int(input('please enter col: '))
    set_cell(board, row, col, marker)
# [...]

```

Listing 4.2: version of `get_move()` which ensures that row and col are valid.

In my experience students can have trouble understanding that they have an editor can change the existing function (thus typing very little), while I have to write a lot more on the board. I have seen the students type the whole thing again, ending up with two copies of the the function, so it is worth shepherding them through this first significant edit.

We are left with one final problem: the current program allows you to stomp on an existing cell, so we need to modify `get_move()` to check for that:

```
# [...]
def get_move(board, marker):
    """asks the player for a move; player types numbers 0, 1 or 2 for row
    and col"""
    valid = False
    while not valid:
        row, col = -1, -1
        while not row in [0, 1, 2]:
            row = int(input('row? '))
        while not col in [0, 1, 2]:
            col = int(input('col? '))
        if board [row][col] == ' ':
            valid = True
    set_cell(board, row, col, marker)
# [...]
```

Listing 4.3: version of `get_move()` which checks if the cell is free

When I write this last function on the board I start paying attention to indentation and really pointing out what the levels of loops/logic are. I will usually take a different color chalk or marker and draw arrows with a head for each level of indentation, writing 4, 8, 12, ... This is important because students will often make mistakes in the indentation of their code.

Try it out! See if you can set row or column to be less than 0 or greater than 2. See if you can stomp on existing cells.

Before moving on to detecting a winner, let us clean up the flow of the main program. Our students should have commented on how the main function has a lot of repetition in taking moves from the players, which is quite unnecessary. We will use a *while loop* to clean that up, and our while loop will for now just run forever:

```
# [...]
def main():
```

```
board = new_board()
print_board(board)
## start a loop that takes moves from the players
## NOTE: you need to type control-C to exit the program
while True:
    get_move(board, 'x')
    print_board(board)
    # move to next player
    get_move(board, 'o')
    print_board(board)
# [...]
```

Listing 4.4: ttt.py - main function cleaned up with a while loop

This is a good time to point out how short and sweet this main function is: it *tells the story*. We can also explain how the terminal interrupt character (control-C) can help them exit this program when it seems to go on forever.

But while pointing out that this is sweet, we can also point out that we have done no more than re-invent the paper and pen: we have created ancient Egyptian technology, and we should feel a burning desire to move beyond.

Lesson 5

Tic-tac-toe: simplest computer play

At this point I ask the class what the program most needs. The kids might suggest the two most important things:

- * Determine if there is a winner.
- * Have the computer play some moves.

I encourage the kids to start with the second one: programming the simplest computer algorithms. This is a good idea because checking for a winner is a bit tedious, so it's a morale booster to see the computer playing against us.

Still, if the class is really determined, you could first jump to Chapter 6 and then come back here.

5.1 First found

First we write a very simple algorithm that puts a marker in the first empty cell it can find.

Here I sometimes will draw the board and tell the class something like “when I come up with an algorithm I like to play a movie in my mind of what it's going to look like” and then I demonstrate visually how the first found algorithm will walk through the various rows and the cells in each row.

Then we start typing!

We write a function `play_computer_first_found()`:

```
def play_computer_first_found(board, marker):  
    """simplest computer algorithm: put your marker on the first empty
```

```
cell you find"""
for row in range(3):
    for col in range(3):
        if board[row][col]==' ':
            set_cell(board, row, col, marker)
            return
```

Listing 5.1: play_computer_first_found()

We also modify the main loop to play the computer's move instead of getting a human move:

```
# [...]
def main():
    board = new_board()
    print_board(board)
    while True:
        ## ask the player for a move
        get_move(board, 'x')
        print_board(board)
        ## now play the computer move
        play_computer_first_found(board, 'o')
        print_board(board)
# [...]
```

Listing 5.2: ttt.py - with computer playing moves

This will allow students to play a game against the computer. They will enjoy it enormously until they realize that the computer is making very uninteresting moves. This will be fixed in further sections.

The students should be able to quickly devise a strategy to beat the computer every time.

5.2 Random play

Next we will write an algorithm to play a random move for the computer. We start by showing how Python can generate random numbers:

```
>>> import random
>>> random.random()
>>> random.random()
>>> random.random()
>>> random.randint(-3, 10)
>>> random.randint(-3, 10)
>>> random.randint(-3, 10)
>>> random.randint(0, 2)
>>> random.randint(0, 2)
```

```
>>> random.randint(0, 2)
>>> random.randint(0, 2)
>>> random.randint(0, 2)
```

The students can keep repeating these calls with the arrow keys until they are convinced that they have a random sequence. This is also a good time to discuss what random means. As usual, these more “background” and “philosophical” discussions should eventually be truncated so that we can continue the hands-on work.

Next we have a discussion with the students, asking each student to volunteer ideas on how to have the program place a marker at random.

Since we will be using the random number library, as well as the system exit function, at the top of our program we put the lines:

```
#!/usr/bin/env python3

import random
# [...]
```

Listing 5.3: import statements at top of file

Then we we write our `play_computer_random()` routine:

```
def play_computer_random(board, marker):
    """another simple computer algorithm: place your marker in a random
    empty location"""
    while True:
        row=random.randint(0, 2)
        col=random.randint(0, 2)
        if board[row][col]==' ':
            set_cell(board, row, col, marker)
            return
```

Listing 5.4: `play_computer_random()`

```
# [...]
    ## now play the computer move
    ## uncomment the line for the algorithm you want to use
    #play_computer_first_found(board, 'o')
    play_computer_random(board, 'o')
    print_board(board)
# [...]
```

Listing 5.5: `main()` - adapted to allow choice of computer algorithm

The students can now to play more games against the computer. They might find it slightly more challenging than playing against the `first_found` algorithm.

And if there is a bit of time and you are feeling impish, you can suggest that they have the computer play itself: first-found algorithm against random algorithm!

Lesson 6

Checking if someone has won

There are many areas of improvement for the entire program. In Chapter 7 we will add the very important functionality of having the computer play a more interesting game. But before we get to that we must add some important smarts: having the computer check if there is a winner, and hence quit at the right time.

The condition for winning tic-tac-toe is “three in a row”: you need three of the same symbols in a column or a row or one of the two diagonals.

We will write functions that return `True` or `False` if a row has three ‘x’ or three ‘o’ cells. Rows are the easiest since we represent our board as a list of rows, but we will be able to use this function for columns and diagonals as well, though we will need to *extract* them from the board.

```
def find_winner(board):
    """return winner 'x' or 'o', or ' ' if there is no winner yet"""
    for row in board:
        winner = find_winner_row(row)
        if winner != ' ':
            return winner
    for col_no in range(3):
        col = extract_col(board, col_no)
        winner = find_winner_row(col)
        if winner != ' ':
            return winner
    winner = find_winner_row(extract_slash(board))
    if winner != ' ':
        return winner
    winner = find_winner_row(extract_bslash(board))
    if winner != ' ':
        return winner
    return ' '
```

Listing 6.1: function that tests if there is a winner

As often happens, this function requires four other functions to be written: `find_winner_row()`, `extract_col()`, `extract_slash()` and `extract_bslash()`.

Here they come. This is a marathon portion of the course (writing a ton of new code in a hurry), but quite worth it.

```
def find_winner_row(row):
    """sees if the given row has a solid win. note that a "row" is just a
    list of 3 cells, so if you pack a column or diagonal into this
    list it will also work to find column and diagonal winners
    """
    if row[0] == row[1] and row[1] == row[2]:
        return row[0]
    else:
        return ' '

def extract_col(board, col_no):
    """packages the entries for the given column into a list"""
    col = [board[0][col_no], board[1][col_no], board[2][col_no]]
    return col

def extract_slash(board):
    """packages the entries for the "slash" diagonal into a list"""
    slash = [board[0][2], board[1][1], board[2][0]]
    return slash

def extract_bslash(board):
    """packages the entries for the "backslash" diagonal into a list"""
    bslash=[board[0][0], board[1][1], board[2][2]]
    return bslash
```

Listing 6.2: functions needed for `find_winner()`

And finally, to resolve the issue of ending a game, we also need a function to see if the board is full:

```
def board_is_full(board):
    """returns True if the board is full, False otherwise"""
    for row in board:
        for cell in row:
            if cell==' ':
                return False
    return True
```

Listing 6.3: `ttt.py` see if we have a winner in `main()`

Now we can modify `main()` to use this new code and see if there is a winner.

```
# [...]
def main():
    board = new_board()
    print_board(board)
    while True:
        ## ask the player for a move; note that this line could be
        ## replaced by one of the "play_computer..." functions below
        get_move(board, 'x')
        print_board(board)
        winner = find_winner(board)
        if winner != ' ' or board_is_full(board):
            break
        get_move(board, 'o')
        print_board(board)
        winner = play_computer_random(board)
        if winner != ' ' or board_is_full(board):
            break
    if winner == ' ':
        print('Tie!')
    else:
        print('winner is', winner)
# [...]
```

Listing 6.4: `ttt.py` see if we have a winner in `main()`

And we see a big leap forward in our program: it actually has an algorithm that makes a decision! If you play against the computer it should report victories and ties, and exit.

We can now point out that our program has finally gone even further beyond acting like paper and pencil: we now have a smart piece of paper which announces the winner.

Lesson 7

Tic-tac-toe: more intelligent-play

Final stretch: making our computer with some intelligence.

7.1 Defensive play

Now let us add another computer-playing algorithm called `play_computer_defensive()`. This will check to see if the opponent is about to win.

If the opponent is about to win, the computer needs to block it by placing its own marker on the last available slot on the opponent's winning slot.

We start with a description of the algorithm (procedure):

1. Iterate through the three rows one by one.
2. For that row we go through each cell.
3. If that cell is blank then we play a *hypothetical* opponent move in that cell.
4. See if that hypothetical move lets our opponent win.
5. Restore the board by returning that cell to a blank (space).
6. If the opponent could win on that square, we put *our own* marker on that square to block them.
7. If we have no defensive moves to play, we default to playing the random algorithm.

We will need some “helper functions” to reflect the steps of the algorithm:

```
def find_win_candidate(board, marker):
    """scan each cell on the board to see if the given marker could
    win on that cell. returns a triplet (row, col, marker) if that
    marker could win; returns (None, None, None) if there is no
    victory chance"""
    for row in range(3):
        for col in range(3):
            if board[row][col] == ' ':
                is_win = check_if_winning_move(board, row, col, marker)
                if is_win == marker:
                    return [row, col, is_win]
    return [None, None, None]

def check_if_winning_move(board, row, col, marker):
    """See if the given cell allows a win by the given marker. Returns
    that winning marker if so, otherwise returns a space ' ' """
    assert(board[row][col] == ' ') # only call this on blank cells
    ## temporarily set the cell to this marker
    set_cell(board, row, col, marker)
    winner = find_winner(board)
    ## restore the cell to a space
    set_cell(board, row, col, ' ')
    ## see if there is a winner
    if winner == marker:
        return winner
    return ' '

```

Listing 7.1: find_win_candidate() - functions to see if someone is about to win

Then we write play_computer_defensive() to use those helper functions:

```
def play_computer_defensive(board, my_marker):
    """plays a defensive strategy: if there is a threat by the opponent,
    we plug it up; otherwise we play the random algorithm"""
    opp_marker = 'x' if my_marker == 'o' else 'o'
    [row, col, win_candidate] = find_win_candidate(board, opp_marker)
    if win_candidate == opp_marker:
        set_cell(board, row, col, my_marker)
    else:
        play_computer_random(board, my_marker)

```

Listing 7.2: play_computer_defensive()

Finally we set call our new function in main():

```
# [...]
    ## uncomment the line for the algorithm you want to use

```

```

    #play_computer_first_found(board, marker)
    #play_computer_random(board, marker)
    play_computer_defensive(board, marker)
# [...]

```

This is a good place to comment on (a) how short and easy it was to write this function once we had already written all the other helper functions, and (b) how easy it is to read this function.

7.2 Opportunistic play

Now let us add a computer-playing algorithm called `play_computer_opportunistic()`. This will check to see if there is a cell where putting our marker would win!

We can use these helper functions from before, using the our own marker instead of the opponent's to see if *we* are about to win. Note that if we don't have the opportunity to win we back up to playing the defensive algorithm.

```

def play_computer_opportunistic(board, marker):
    """plays opportunistically: if we can win we put our marker in that
    slot; otherwise we play the defensive algorithm"""
    [row, col, win_candidate] = find_win_candidate(board, marker)
    if win_candidate == marker:
        set_cell(board, row, col, marker)
    else:
        play_computer_defensive(board, marker)

```

Listing 7.3: `play_computer_opportunistic()`

Finally we set call our new function in `main()`:

```

# [...]
    ## uncomment the line for the algorithm you want to use
    #play_computer_first_found(board, marker)
    #play_computer_random(board, marker)
    #play_computer_defensive(board, marker)
    play_computer_opportunistic(board, marker)
# [...]

```

7.3 Concluding words

The best computer-playing algorithm is much more complex and it involves some restructuring of the program. At this point I leave it as an exercise for the students to do with a mentor.

The problem is this: to have the computer play an optimal move by looking ahead at what the opponent might do.

Based on this write a function called `play_computer_lookahead(board, marker)` which plays the optimal game of tic-tac-toe.

And with that encouragement to move forward, we can conclude class by having the students play the various algorithms (first_found, random, opportunistic, defensive) *against each other!*

And they should try to predict which algorithm will win.

And as their parents come pick them up, they should try having `play_computer_opportunistic()` play against their parents.

Appendix A

Life after the course

It is important to let the children and parents know that they can correspond with you after the course. That way they can continue working on the program and ask questions if they get stuck.

I have taken the approach of also offering in-person help after the course by pointing families to the New Mexico GNU/Linux user group meeting. I go to many of those meetings, as do other hackers, and we are always willing to help.

I have also developed a “scientific computing for kids” course which requires nothing but this course’s material. I have written a teacher’s manual for it (Galassi 2016), analogous to this one.

Another continuation would be some lessons on GUI programming, in particular developing a GUI for this tic-tac-toe game.

Appendix B

Notes on installing GNU/Linux in class

B.1 Issues with old/cheap hardware

An old computer can still be quite useful: the GNU/Linux operating system is often much more efficient with hardware resources, so it can give new life to an old hunk of metal, as long as it is less than about 10 years old. (Computers more than 10 years old will usually need a more specialized operating system distribution.)

Still, older computers can have some problems. Mostly these will come down to:

limited memory This is the biggest problem. Today's (early 2015) desktops require more than 2GB of RAM or they will not operate very well. You can install a low memory use distribution, such as LUbuntu, but the problem really comes down to the web browser: a contemporary version of full featured browsers like Firefox and Chromium will rapidly climb up to using several gigabytes of RAM, slowing down your computer dramatically. On GNU/Linux systems it is easy to install the Midori web browser which works on many "fancy" web pages but does not use too much memory.

limited disk space An old computer might come with a small hard drive. It turns out that this is seldom a problem: computers less than 10 years old usually have hard disks with more than 5GB of storage, and that is good enough for our purposes.

slower CPU This is actually not really a problem: the limited memory will slow you down much more than an older CPU.

older graphics card Older graphics cards don't offer some of the fancy visual effects that are used in contemporary desktops, such as "heads up display" features. This is not really a limitation: there are dozens of choices of window manager on a GNU/Linux system, and many hackers prefer the non-compositing window managers which run well on older graphics cards.

A note on very low RAM: the world of low-memory GNU/Linux distributions is a messy one, with many of them being poorly maintained or fiddly to use. In spring 2016 I evaluated various alternatives and found Bodhi to be more usable than the others, but apparently Simplicity Linux is also currently maintained.

B.2 The Asus X551MA laptop

This is a very inexpensive laptop which offers rather light weight, big screen, big hard drive (half a terabyte) and plenty of RAM (4GB). The low price is probably due to the lack of certain features like bluetooth and a DVD drive, which does not affect software development at all.

B.2.1 Preparing a USB memory stick

This laptop has 4GB of RAM so we can comfortably run a full contemporary version of a GNU/Linux distribution. In my course I just point people to the Ubuntu GNU/Linux distribution.

Download the Ubuntu 14.04 image from <http://www.ubuntu.com/download/desktop>

Then take a 2GB (or bigger) memory stick and prepare it on an existing computer following the instructions at the following URLs, depending on whether you are coming from:

GNU/Linux <http://www.ubuntu.com/download/desktop/create-a-usb-stick-on-ubuntu>

MS Windows <http://www.ubuntu.com/download/desktop/create-a-usb-stick-on-windows>

MacOS <http://www.ubuntu.com/download/desktop/create-a-usb-stick-on-mac-osx>

B.2.2 Saving off laptop info and restore drive

It might be a good idea to save *another* USB memory stick with the computer's Windows installation. This is called a "recovery drive". This is not because we might want to use Windows (we don't), but rather in case we should need to return the computer and thus need it to have its original operating system.

1. Write down the PC product key. In Windows do **Settings** -> **PC And Devices** -> **PC Info** and you will see the product key.
2. Make a recovery drive on a 16GB USB memory stick. Note that the memory stick will be erased. Plug it in and go to **Settings** -> **Control Panel** -> **Action Center** -> **Recovery** -> **Create recovery drive**

B.2.3 Booting from a USB stick

Recent vintage computers have a "secure boot" feature which is aimed at making it harder to replace Microsoft Windows on that computer. I have found that these Asus computers allow you to boot from a memory stick without problems.

Follow this procedure:

1. Insert the USB drive with Ubuntu GNU/Linux 14.04
2. Power on the laptop while holding the **ESC** key
3. You will get a "boot menu" which allows you to either
 - boot Windows
 - boot the Ubuntu GNU/Linux installation USB memory stick
 - enter a setup menu

Choose to boot from the USB memory stick.

B.2.4 Installing GNU/Linux

You can select "run Ubuntu without installing" and it will put you into a working desktop environment.

Here you will have an install icon. You should double-click that icon and it will start the installation procedure.

You can answer most of the questions in the obvious default manner and it will go well for you.

One place where you might want to change from the default is when it asks if you want to “install 3rd party software”. Turning this on can install useful software to play mp3 files and view movies.

One question will be if you want to “download updates while installing”. In a classroom setting I would *not* set that option because it can take much more time to install. It is quite easy to run:

```
sudo apt-get update
sudo apt-get dist-upgrade
```

later on.

The trickiest question is about partitioning the hard drive. For our purposes you should erase all existing partitions and install GNU/Linux on the entire drive.

At this point your computer will reboot into the GNU/Linux operating system and you are ready to go.

B.3 The old Mac G4 PowerPC laptop

These are older, somewhat chunky Apple laptops with a PowerPC processor instead of an Intel Pentium. Many of these are still around. They are pretty much unusable with the Mac operating system, which has not been supported on the PowerPC since 2007, but can still be used with various GNU/Linux distributions.

I experimented with some old G4 laptops and various distributions and came up with the following recipe. In your searches you will find the Mint PPC distribution, which I found to take many many hours to install, and the Ubuntu 12.04 PPC, which I found installed in a reasonable amount of time. There will certainly be many others as well, but here I will show how to install the lightweight Ubuntu 12.04 PPC distribution.

B.3.1 Preparing an installation CD

You can download an LUbuntu 12.04 PowerPC image from <https://wiki.ubuntu.com/PowerPCDownloads> which will take you to <http://cdimage.ubuntu.com/lubuntu/releases/precise/release>

Pick the image with a name like `lubuntu-12.04-alternate-powerpc.iso` and download it. Once you have that ISO file you burn it onto a blank CD.

B.3.2 Booting from CD and installing

To boot a G4 laptop from the CD you need to:

1. insert the CD
2. power on the computer while holding down the ‘c’ key

(note that you might need to power the computer on enough to get it to swallow the CD, and then power it off and turn it on again so you can hold the ‘c’ key and boot from CD.)

You are now given the “yaboot prompt” that says `boot:` and you can just hit the `enter` key to start booting.

The installation procedure asks you many questions. You can give the default answer for most of them, but a few will require a specific response from you:

keyboard You could look for the Mac keyboard variant, but it will work fine with the default American keyboard.

network It might not find wi-fi immediately and we might need to install special wi-fi drivers once the whole system is up. It is OK to install without a network, or even better to plug it in to a wired ethernet network.

time zone You should put in your time zone. Here in Santa Fe it will be US Mountain time.

disk partitioning Choose Guided, use entire disk.

user account Create yourself a login name. I recommend that you pick something that is all lower case. You can use letters, digits, underscore and hyphen, but absolutely never use a space in your login name. Mine, for example, is `markgalassi`

Of course for your full name you should include a space!

B.3.3 Post installation

You are almost ready to go with a wired ethernet network, but we will take one last step to get the wi-fi working. We will install the package `firmware-b43-installer` with the following instructions:

```
sudo apt-get update
sudo apt-get firmware-b43-installer
```

At this point, possibly with a final reboot, you should be ready to go with wi-fi and all.

But these are truly old computers with little memory, so you might install the Midori browser with:

```
sudo apt-get install midori
```

and to use that instead of Firefox if you are always low on RAM.

Appendix C

Software Freedom

A big part of what has made my scientific career pleasant and effective has been the free software movement. Using software for scientific research, without the burden of being hostage to a corporation's marketing plan, is very liberating.

In the 1980s Richard Stallman founded the GNU project, aimed at providing a complete free (as in freedom) high quality computing environment.

The operating system and each individual program is offered to users with full freedom to use, modify and customize (through access to source code) and redistribute (including redistributing modified copies).

Many thousands of volunteers and paid professionals have contributed to the free software movement to the point where the machines running the GNU/Linux operating system are now the most important part of the infrastructure of the computer world.

It is in that spirit that I teach these classes: I want students to learn very serious programming, but I also want them to know that we can develop software not just to make a widget for a company to sell, but also because it can be part of a large effort to improve the high tech world.

In particular, this book is a free (as in freedom, but also free in cost) manual for anyone who might want to teach similar courses.

This book can be redistributed under the terms of the GNU Free Documentation License (see Appendix D).

Appendix D

GNU Free Documentation License

Version 1.2, November 2002
Copyright © 2000,2001,2002 Free Software
Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301
USA

Everyone is permitted to copy and distribute
verbatim copies of this license document, but
changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a worldwide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “**Document**”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “**you**”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “**Modified Version**” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “**Secondary Section**” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “**Invariant Sections**” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is

suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “**Entitled XYZ**” means a named sub-unit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “**Entitled XYZ**” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties; any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your

Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with ... Texts.” line with this:

with the Invariant Sections being
LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and
with the Back-Cover Texts being
LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.