1. Introduction

This is a course in problem-solving using a computer. Writing down a set of steps or instructions to make a computer perform some task for you is called programming. We’ll see that there are two parts to this process. Given a problem to solve, we have to

1) figure out what the steps are in solving the problem, and then
2) write them down in such a way that a computer can interpret and carry them out.

You might be concerned about the second part. How do you talk to a computer? But that turns out to be relatively easy. We just need a programming language that our computer can interpret. In this course, we will be using a language called Python. Python is a good choice because it is very simple to start using. (However, keep in mind that this is not an in-depth course in Python; we will really only be using a small fraction of the features of the language.)

We will soon discover that the first part – that is, figuring out exactly what steps are involved in solving a problem – is actually much harder than writing instructions in a programming language. Here we will need a pencil and paper and some clear thinking.

1.1. People are good at solving problems

It isn’t that solving problems is difficult. In fact, it is precisely the opposite: people are so good at solving problems, most of the time we’re not aware of how we’re doing it! For example, think about some of the things you do every day that involve some kind of problem-solving strategy:

Making a peanut butter and jelly sandwich
Finding a parking spot
Arranging a time for three friends to meet
Getting a good deal on a phone
Driving across town

Now, suppose you had to spell out all the steps and little decisions you had to make in order to do one of these tasks. For example, what’s really involved in making a peanut-butter-and-jelly sandwich?

Is there bread? Check if the bread is moldy. Find the peanut butter. Remove the lid. If the jar is empty, find another jar. Remove the lid and then the seal from the jar. Find a knife. If there are no knives in the drawer, get a dirty one from the sink and wash it. Use the knife to spread peanut butter on one piece of bread.
Programming a computer is a bit like this. You really have to spell out every step of the process, because computers can only perform very simple steps.

1.2. Describing a problem-solving strategy

Of course, without some fancy robotic arms we certainly aren’t going to program a computer to make sandwiches for us. But here’s a much more straightforward example we can think about. Suppose you have a list of numbers, like this for example:

43 17 85 32 86 79 18

What’s the biggest number in the list? Pretty easy, right? You can just spot the biggest one without even thinking. But what if you had a longer list, maybe like this:

47 26 20 4 60 70 8 24 33 58 20 83 53 95 37 67 85 93 83 49 79 83 61 79 48 28 97 77 89 45 43 41 44 47 31 71 52 22 62 2 82 92 50 1 58 5 26 64 87 82 18 45 11 31 35 59 78 96 91 14 3 65 14 15 94 4 31 41 16 11 43 9 87 1 94 80 2 24 5 21 60 10 97 80 69 61 65 16 89 17 68 77 3 36 50 48 81 6

Well, you might have to be a bit more systematic. Go ahead and find the biggest number, and then ask yourself how you did it.

Most of us end up doing something like this:

1. Look at the first number, and remember it (that’s our maximum so far)

2. Read through the rows from left to right

3. If we’ve run out of numbers, then we’re done.

4. Otherwise, look at the next number and compare it to the maximum we remembered

5. If the new number is bigger, then remember that one instead

6. Go back to step 3

The sequence of instructions above gives us a strategy, also called an algorithm, for solving the problem of finding the biggest number in a list. It is a bit more wordy than just saying “find the biggest number in the list”. But that’s how it works: you have to literally write down every step.
How do you know when the steps are clear enough? One way to think about how a computer works is that it’s like talking to a kid, say, a reasonably bright fourth-grader. She can read and do fractions and follow directions, but she doesn’t necessarily have any life experience and doesn’t have the “big picture” of what you are trying to accomplish. If you can write down your instructions clearly enough so that a reasonably bright fourth-grader can carry them out, then chances are you’ll be able to program them for a computer too.

(picture of 4th-grader in box with clipboard and pocket calculator)

The idea of a variable

One thing to think about here is: what does it mean to “remember” a value, as in step 4 of our strategy above? In programming we need some way to store values and recall them later. We use variables for this purpose. In programming, a variable is a bit like the variables you’ve seen in math books, for example, if someone writes:

\[
x = 42 \\
y = 2x + 1
\]

you would probably agree that \(y\) is now 85. That is, in the second line you recognize that \(x\) still has the value 42. There’s just one important difference between variables in math books and variables in programming. In programming, the value of a variable can change as the steps are executed. A variable works just like the “memory” key on a pocket calculator.

One way to think of a variable is that it is like a page on the clipboard our fourth-grader is holding. She can write down a number, and look it up later, but she can also erase it and write a new number if you ask her to.

(A variable - picture of clipboard labeled ‘max’)

1.3. Picturing a problem-solving strategy using a flowchart

We described our strategy for finding the biggest number as a sequence of written instructions. There is a pictorial way to describe the strategy that will be useful to us, called a flowchart. Sometimes a flowchart is more clear than a sequence of written instructions. You can trace through the steps by following the arrows with your finger. In the flowchart, we’re assuming we have a variable called “max” in which we always store the largest value we’ve seen so far. Each time we find a larger value, we have to update the variable “max.”
To make sense out of the “flowchart” picture, let's just try it for a simple list like this:

17  4  137  42

Start out with max equal to 17.
Are there more numbers?
Yes, the next one is 4.
Is 4 bigger than 17?
No, so do nothing.
Follow the arrow back to the top.
Are there more numbers?
Yes, the next one is 137.
Is 137 bigger than 17?
Yes, so change the value of max to 137
Follow the arrow back to the top.
Are there more numbers?
Yes, the next one is 42.
Is 42 bigger than 137?
No, so do nothing.
Follow the arrow back to the top.
Are there more numbers?
No, so the result is the value of max, or 137.

1.4. The big picture

Before we break for today let’s take a step back and look at the big picture. What are the ingredients that went into the strategy we described above? Here’s what we needed to be able to do:

- store a value so we can remember it later
- do basic arithmetic (like comparing two numbers)
- check a condition and do something or not, depending on whether the condition is true
- repeat some action, continuing as long as some condition is true
- get input or produce output (in order to read the list and report the result)

The surprising thing is that those five ingredients are enough to any computation. In fact, that is all that any computer ever does!

So you can see that the difficulty is programming a computer isn’t that computers are “smart” or “complicated”. The difficulty is that they’re so incredibly stupid that we have spell everything out in detail! The challenge in learning to program is that we have to take our wonderful human problem-solving skills and slow ourselves down enough to analyze how we’re solving a problem, so we can describe the process in simple steps.
We have not talked about the second part of the problem-solving process: how to write down the problem-solving strategy so that a computer can do it for us. That’s coming next!
2. Introducing Python and the shell

In the last unit we brought up the idea that using a computer to solve a problem really has two aspects:

1) figuring out what the steps are in solving the problem, and then
2) writing them down in such a way that a computer can interpret and carry them out.

Last time, as an example of the first aspect, we used the problem of finding the largest number in a list. Remember that we came up with a strategy for solving the problem, and wrote down the steps of the strategy as a sequence of instructions to follow. We also represented the same strategy as a picture called a flowchart. We also made the observation that human beings are really good at solving problems – so good, in fact, that it is sometimes hard for us to slow down and analyze how we’re solving the problem. Today we want to look at the second aspect, and start learning how to write down the steps of a problem-solving strategy so that a computer can execute them.

2.1. It’s hard to talk to a machine

It turns out that human beings are also really, really good at language and communication. You can say all kinds of completely ambiguous things, you can use double meanings, puns, sarcasm, and allusions, and other people will usually know what you’re talking about. For example,

“Hey, bring me that thing on the table.”
“Life is like a box of chocolates.”
“The spirit is willing, but the flesh is weak.”
“Your teeth are like stars; they come out at night.”

You kin mispell woords and leave out punctuation You, can, put, in, too, many, commas, and people will still be able to read your writing.

But when you’re talking to a computer, it is just the opposite. With a computer you have to use a programming language with very precise structure. You have to get every detail of the grammar and punctuation and spelling just right, and you can’t have any ambiguity at all. Have we mentioned before that computers are really dense?

2.2. A first look at Python

Like most computer languages, Python has the following ingredients:
keywords (such as the \texttt{print} keyword used below) 
operators (such as +, *, <, etc.) 
literal values (such as 42, 3.14, “Hello”) 
identifiers (such as variables and function names) 
syntax rules (grammar and punctuation)

Statements you write in Python are not directly executed by your computer’s hardware, rather, they are executed by an application called the Python \textit{interpreter}. One nice thing about the interpreter is that it comes with an \textit{interactive shell}, where you can easily experiment with the effects of Python statements, and the values of expressions.

There are many ways of starting a Python shell. We will suggest some in the lab exercises. But in all cases, when you start the shell you’ll see the \texttt{prompt “>>>”}. This means the shell is ready for you to type something. Here are some things to try:

```python
>>> print 42 + 5
47
```

Let’s figure out what’s going on here.

- The whole thing, \texttt{print 42 + 5}, is a \textit{statement}. A statement is an instruction to do something. In this case, the print statement is telling the interpreter that it should display something on the screen.

- \texttt{42 + 5} is an \textit{expression}. An expression represents a value. In this case, the value is the number 47. Notice that \texttt{42 + 5} itself has been formed by \textit{composing} two simpler expressions, 42 and 5, which are \textit{literal} values.

- \texttt{print} is a Python keyword

Since we are using the interactive shell, the result of executing the print statement is displayed as soon as we type it.

We could also display a string of text, surrounded by double or single quotes.

```python
>>> print "Hello"
Hello
```
>>> print 'Hello'
Hello

One feature of the shell is that if you just type an expression, it will assume you want to display its value, even if you don’t write a print statement. This makes it easy to experiment with expressions:

```python
>>> 42 + 5
47
>>> "Hello"
'Hello'
```

(Notice the way ‘Hello’ is displayed with quotes. You can see that the way a value is displayed may be slightly different from the way it appears using the print statement.)

### 2.3. Types of data

One thing that becomes important to us very quickly is that every value has a type. What does that mean? Well, here’s the basic idea: a computer is nothing but a whole bunch of tiny electrical switches. A switch can be “on” or “off”, and we often think of “on” as the number 1 and “off” as the number 0. Somehow those ones and zeros have to be interpreted as meaningful values like numbers and text. In order to know how it should interpret a given bunch of ones and zeros, the system needs to know what kind of value it is supposed to be - a number, text, a part of a picture, or some other type of value.

We can find out the type of a value using a built-in function called type:

```python
>>> type(42 + 5)
<type 'int'>
>>> type("Hello")
<type 'str'>
>>> type(3.14)
<type 'float'>
```

Type ‘int’ means “integer” (positive and negative whole numbers)
Type ‘str’ means “string of text”
Type ‘float’ means “floating-point number”, that is, numbers with a decimal point

What’s the difference between int and float? Aren’t they both just numbers? Computers have to distinguish between whole numbers (the type int in Python) and numbers with a decimal or fractional part (the type float in Python). For numeric literals, the interpreter normally deduces the type to be float if you enter the value with a decimal point.
Introducing Python and the shell

```python
>>> type(42)
<type 'int'>
>>> type(42.0)
<type 'float'>
```

What about

```python
>>> type("42 + 5")
<type 'str'>
```

Well, anything with quotes around it is just a string of text. This text happens to consist of some digits and a plus sign which could be interpreted as the value 47. But since we put it in quotes, all the interpreter sees is that it is a literal string of six characters.

```python
>>> print 42 + 5
47
>>> print "42 + 5"
42 + 5
```

### 2.4. Syntax errors

Now, not everything we enter into the shell is going to work:

```python
>>> 42 5 +
File "<stdin>", line 1
  42 5 +
^  
SyntaxError: invalid syntax
```

This is called a *syntax error*, meaning that we didn’t follow the exact grammar for Python expressions. You’ll get a similar error if you misspell a word or use the incorrect case (as in most programming languages, everything in Python is case-sensitive). It’s a good idea to try making errors on purpose to see what happens. Then, when you make errors by accident, you’ll have some idea what’s going on.

For example, what if we forget to type the ending quotation marks?

```python
>>> "Hello
File "<stdin>", line 1
  "Hello
^  
SyntaxError: EOL while scanning string literal
```

© Steven M. Kautz 2009-2011
What if we misspell the print keyword?

```python
>>> rpint "Hello"
File "<stdin>", line 1
    rpint "Hello"
^  
SyntaxError: invalid syntax
```

What if we accidentally capitalize the keyword?

```python
>>> Print "Hello"
File "<stdin>", line 1
    Print "Hello"
^  
SyntaxError: invalid syntax
```

As you can see, the error messages don’t always do a very good job of explaining what’s going wrong! Here is a useful tip to remember: when you get an error message in the shell, start reading it at the bottom. Usually the last line in the error message gives you the best clue. Unfortunately, the interpreter doesn’t always “know” what went wrong, it just reports the first place it got stuck trying to interpret what you typed.

### 2.5. Doing arithmetic

Expressions can be composed from other expressions using the arithmetic operators. These are the ones you are familiar with, except that a star is used for multiplication instead of a dot or “x”.

+ add
- subtract
* multiply
/ divide

You can also use parentheses to change the order of operations:

```python
>>> (2 * 3) + 4
10
>>> 2 * (3 + 4)
14
>>> 25 / 10
2
```

What about that last one? Shouldn’t we get 2.5? The behavior of the division operator might be surprising. When it is used for whole numbers, the interpreter performs integer division, which is like the kind of division you used to do in grade school:
“25 divided by 10 is 2, with 5 left over”.

So that’s where we get the answer 2. If what you really want is the remainder, there is a special operator for that. The percent sign “%” is called the modulus operator. We read this as “25 mod 10”. It just means “the remainder when 25 is divided by 10”.

```python
>>> 25 % 10
5
```

If one or both of the numbers has type float, the interpreter will perform a floating-point division like your pocket calculator:

```python
>>> 25 / 10.0
2.5
```

There is one more operator we should know about, which is for raising a number to a power. Something like two to the power 5 ($2^5$) is written $2^{**}5$.

```python
>>> 2 ** 5
32
```

### 2.6. Writing a script

The shell is useful for experimentation, but if you want to do anything useful involving more than a couple of statements, you don’t want to have to retype them in the shell every time. Fortunately, we can type up Python statements and save them in a file to use later. Such a file is called a script or program.

Here is an example:

```python
# This is my first Python script
print 2 + 3
print "Hello, world!"
```

The statements in the script aren’t executed until we tell the Python interpreter to run the script. When we run this script, we see the output:
Hello, world!

Notice that first line starting with the pound sign “#” doesn’t seem to be having any effect when the script runs. It is called a comment. Comments are ignored by the interpreter but are useful for us, because they can help explain what the script (or part of the script) is for.

In the shell, you can just type an expression like 2 + 3, and the interpreter will evaluate it and display the value immediately. That’s just the way the shell works. But if you think about it, 2 + 3 really isn’t an instruction to “do” something with the value 5, it just “is” the value 5. So in a script, writing an expression by itself has no effect. For example, when we run the following script, the output is just

Hello, world

We always try to distinguish between expressions and statements in Python.

- A statement is an instruction to do something. So far, the only kind of statement we know about is the print statement.

- An expression represents a value, such as 2 + 3 or “Hello”.

Try adding an extra space before the print keyword and see what happens. Notice the error message: “unexpected indent”. This reveals one distinctive feature about Python: indentation matters. We will see later how indentation is conveniently used to show the structure of more complicated scripts.

There is one other bit of jargon to get used to: programmers refer to anything written in a programming language as “code.”
3. Variables and assignment statements

In unit 1 we saw that one of the key ingredients of problem-solving is being able to store a value using a variable and look it up again later. In Python, you can create a variable just by assigning it a value. Assignment is done with the equals sign “=”; except in Python we don’t call it the equals sign, we call it the assignment operator. For example, we can write:

```python
>>> x = 42 + 5
>>> print x
47
```

You read the first line as “x gets the value 42 + 5”. Afterward x has the value 47.

The first line, \( x = 42 + 5 \), is read as “x gets the value \( 42 + 5 \)”. It is an example of an assignment statement. You can see that after it executes, \( x \) has the value 47. Notice when we read an assignment statement, we are careful not to say “is equal to”. An assignment statement doesn’t check whether the two sides are equal, it changes the value of whatever variable on the left. This takes some getting used to!

A good way to think about assignment is that it is really a two-step process:

1. Step 1: evaluate the expression on the right-hand side
2. Step 2: take its value, and store it in the variable on the left-hand side

It only works right-to-left, so there has to be a variable on the left!

(picture)

As an example, let’s try writing an assignment statement backwards:

```python
>>> 42 = x
File "<stdin>", line 1
SyntaxError: can't assign to literal
```

Here are some more examples of assignment statements.

```python
>>> maximum_speed = x + 23.0
>>> greeting = "Hello"
```
Notice that a variable doesn’t have to be a single letter. (although there are some restrictions that we will explore shortly).

We sometimes draw a “picture” of the effects of an assignment by putting the value of the variable in a box with a label. A picture like this is called a memory map, because in reality the value of each variable is stored in the computer’s memory. After the three assignments above, the memory map would look like this.

```
>>> x = 17
>>> y = x
>>> x = 137
>>> print y
```

Now, after those three statements execute, what is the value of y? Let’s draw a memory map and follow what’s going on. After the assignment x = 17 we have the following (we’ll put a question mark in y’s box to indicate that it hasn’t been defined yet):

```
x   ->   17
y   ->   ?
```
The assignment \( y = x \) follows the two-step process. First we evaluate the expression on the right (\( x \)) which has value 17. Then we take that value, and store it in the variable \( y \). The resulting memory map looks like this:

\[
\begin{align*}
\text{x} & \quad \rightarrow \quad 17 \\
\text{y} & \quad \rightarrow \quad 17
\end{align*}
\]

The next assignment \( x = 137 \) stores the value 137 in \( x \), so \( y \) still has the value 17.

\[
\begin{align*}
\text{x} & \quad \rightarrow \quad 137 \\
\text{y} & \quad \rightarrow \quad 17
\end{align*}
\]

The moral of the story is that the assignment statement \( x = y \) really doesn’t mean “\( x \) is equal to \( y \)”. When we store the value 17 into \( y \), it doesn’t matter where that value came from; it’s just the number 17.

### 3.2. The type of a variable

Like any value, a variable has a type. In Python, the type of a variable is determined by the type of the value it stores, and if we later store a different value, the type of the variable can change. (This is a bit weird if you’ve ever seen a language such as Java or C, in which the type of a variable has to be explicitly defined and can never change.)

```python
>>> x = 42
>>> type(x)
<type 'int'>
>>> x = "boogers"
>>> type(x)
<type 'str'>
```

If you try to use a variable that has never been defined, you’ll get an error:
>>> minimum_speed = foo - 7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foo' is not defined

3.3. Restrictions and conventions for naming variables

We mentioned before that variables don’t have to be single letters, but there are some restrictions. A variable has to be a valid Python identifier, which means that it

- must start with a letter or underscore (not a number),
- must contain only letters, numbers, or underscores (no other spaces or punctuation), and
- can’t be a Python keyword.

There are some conventions too. These are things that the interpreter won’t care about, but that Python programmers find useful.

- Use lowercase letters only
- Use meaningful names for variables that have a particular meaning.
- If there are multiple words, separate them with the underscore character, as in maximum_speed.

3.4. An example using variables

Let’s do an example using variables in a simple calculation. How about a script to calculate the two solutions of a quadratic equation, \( ax^2 + bx + c = 0 \), using the famous quadratic formula:

\[
\frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad \frac{-b - \sqrt{b^2 - 4ac}}{2a},
\]

For the sake of this example, let’s start with some values for a, b, and c.

\[
a = 2 \\
b = -9 \\
c = -5
\]
We notice that the expression in the radical, \( \sqrt{b^2 - 4ac} \), occurs in two places. We can simplify things by just calculating it once and storing the value in a variable. For lack of a better name, we’ll just call it temp. We can calculate the square root by raising the quantity to the power 1/2 or 0.5 (if you don’t remember this from algebra, don’t worry about it).

\[
temp = (b \times b - 4 \times a \times c)^{0.5}
\]

Now we want to write something that represents: \( solution1 = \frac{-b + temp}{2a} \). It is tempting to simply write

\[
solution1 = -b + \frac{temp}{2a}
\]

but this illustrates a couple of problems that come up when you try to take a mathematical formula and write it on one line using a programming language. First of all, we can’t just write \( 2a \) for 2 times a, we have to write \( 2 \times a \):

\[
solution1 = -b + \frac{temp}{2 \times a}
\]

That still won’t work! The problem is that the whole expression, \(-b + temp\), is supposed to be the numerator of a fraction, and \( 2 \times a \) is supposed to be the denominator. So we have to put in parentheses:

\[
solution1 = \frac{-b + temp}{2 \times a}
\]

The entire script looks like this now:

```python
a = 2
b = -9
c = -5
temp = (b * b - 4 * a * c) ** 0.5
solution1 = (-b + temp) / (2 * a)
solution2 = (-b - temp) / (2 * a)
print "First solution:", solution1
print "Second solution:", solution2
```

Well, what did we learn from this?

- We can use a variable (temp) to hold the value of a common subexpression
- For multiplication, we always have to write the star operator

We have to use parentheses for fractions in order to write them on one line