

Lesson 1: Noob

Introduction to Programming

The goal of this course is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is problem solving. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called, "The way of the program".

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

What is a program?

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing an image or playing a video.

The details look different in different languages, but a few basic instructions appear in just about every language:

input: Get data from the keyboard, a file, the network, or some other device.

output: Display data on the screen, save it in a file, send it over the network, etc.

math: Perform basic mathematical operations like addition and multiplication.

conditional execution: Check for certain conditions and run the appropriate code.

repetition: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

Introduction to Python

Python is one of those rare languages which can claim to be both simple and powerful. You will find yourself pleasantly surprised to see how easy it is to concentrate on the solution to the problem rather than the syntax and structure of the language you are programming in.

The official introduction to Python is:

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

I will discuss most of these features in more detail in the next section.

Story behind the name

Guido van Rossum, the creator of the Python language, named the language after the BBC show "Monty Python's Flying Circus". He doesn't particularly like snakes that kill animals for food by winding their long bodies around them and crushing them.

Features of Python

Simple

Python is a simple and minimalistic language. Reading a good Python program feels almost like reading English, although very strict English! This pseudo-code nature of Python is one of its greatest strengths. It allows you to concentrate on the solution to the problem rather than the language itself.

Easy to Learn

As you will see, Python is extremely easy to get started with. Python has an extraordinarily simple syntax, as already mentioned.

Free and Open Source

Python is an example of a *FLOSS* (Free/Libre and Open Source Software). In simple terms, you can freely distribute copies of this software, read its source code, make changes to it, and use pieces of it in new free programs. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python.

High-level Language

When you write programs in Python, you never need to bother about the low-level details such as managing the memory used by your program, etc.

Portable

Due to its open-source nature, Python has been ported to (i.e. changed to make it work on) many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features.

You can use Python on GNU/Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and PocketPC!

You can even use a platform like [Kivy](#) to create games for your computer *and* for iPhone, iPad, and Android.

Interpreted

This requires a bit of explanation.

A program written in a compiled language like C or C++ is converted from the source language i.e. C or C++ into a language that is spoken by your computer (binary code i.e. 0s and 1s) using a compiler with various flags and options. When you run the program, the linker/loader software copies the program from hard disk to memory and starts running it.

Python, on the other hand, does not need compilation to binary. You just *run* the program directly from the source code. Internally, Python converts the source code into an intermediate form called bytecodes and then translates this into the native language of your computer and then runs it. All this, actually, makes using Python much easier since you don't have to worry about compiling the program, making sure that the proper libraries are linked and loaded, etc. This also makes your Python programs much more portable, since you can just copy your Python program onto another computer and it just works!

Object Oriented

Python supports procedure-oriented programming as well as object-oriented programming. In *procedure-oriented* languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In *object-oriented* languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.

Extensible

If you need a critical piece of code to run very fast or want to have some piece of algorithm not to be open, you can code that part of your program in C or C++ and then use it from your Python program.

Embeddable

You can embed Python within your C/C++ programs to give *scripting* capabilities for your program's users.

Extensive Libraries

The Python Standard Library is huge indeed. It can help you do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI, FTP, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interfaces), and other system-dependent stuff. Remember, all this is always available wherever Python is installed. This is called the *Batteries Included* philosophy of Python.

Besides the standard library, there are various other high-quality libraries which you can find at the [Python Package Index](#).

Summary

Python is indeed an exciting and powerful language. It has the right combination of performance and features that make writing programs in Python both fun and easy.

Python 3 versus 2

You can ignore this section if you're not interested in the difference between "Python version 2" and "Python version 3". But please do be aware of which version you are using. This book is written for Python version 3.

Remember that once you have properly understood and learn to use one version, you can easily learn the differences and use the other one. The hard part is learning programming and understanding the basics of Python language itself. That is our goal in this book, and once you have achieved that goal, you can easily use Python 2 or Python 3 depending on your situation.

For details on differences between Python 2 and Python 3, see:

- [The future of Python 2](#)
- [Porting Python 2 Code to Python 3](#)
- [Writing code that runs under both Python2 and 3](#)
- [Supporting Python 3: An in-depth guide](#)

What Programmers Say

You may find it interesting to read what great hackers like ESR have to say about Python:

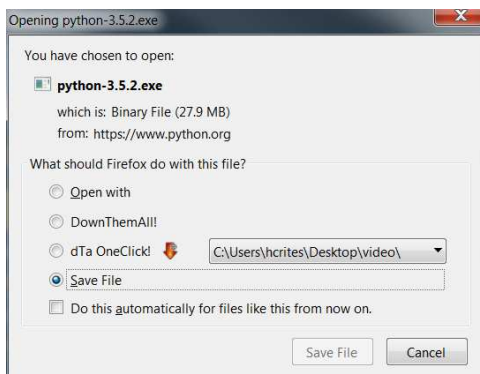
- *Eric S. Raymond* is the author of "The Cathedral and the Bazaar" and is also the person who coined the term *Open Source*. He says that [Python has become his favorite programming language](#). This article was the real inspiration for my first brush with Python.
- *Bruce Eckel* is the author of the famous 'Thinking in Java' and 'Thinking in C++' books. He says that no language has made him more productive than Python. He says that Python is perhaps the only language that focuses on making things easier for the programmer. Read the [complete interview](#) for more details.
- *Peter Norvig* is a well-known Lisp author and Director of Search Quality at Google (thanks to Guido van Rossum for pointing that out). He says that [writing Python is like writing in pseudocode](#). He says that Python has always been an integral part of Google. You can actually verify this statement by looking at the [Google Jobs](#) page which lists Python knowledge as a requirement for software engineers.

Installation

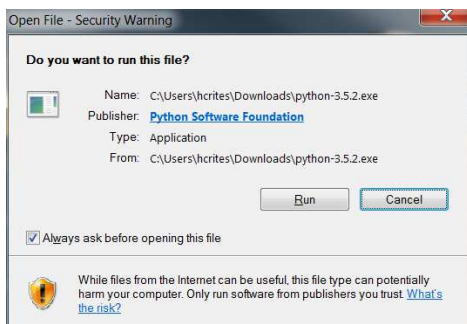
When we refer to "Python 3" in this text, we will be referring to any version of Python equal to or greater than version Python 3.5.2.

Installation on Windows

Visit <https://www.python.org/downloads/> and download the latest version. At the time of this writing, it was Python 3.5.2.



The installation is just like any other Windows-based software.



Make sure you check option Add Python 3.5 to PATH.



You can choose to install Launcher for all users or not, it does not matter much. Launcher is used to switch between different versions of Python installed.

Installation on Mac OS X

For Mac OS X users, use [Homebrew](#): `brew install python3`.

To verify, open the terminal by pressing [Command + Space] keys (to open Spotlight search), type `Terminal` and press [enter] key. Now, run `python3` and ensure there are no errors.

Installation on GNU/Linux

For GNU/Linux users, use your distribution's package manager to install Python 3, e.g. on Debian & Ubuntu:
`sudo apt-get update && sudo apt-get install python3`.

To verify, open the terminal by opening the Terminal application or by pressing Alt + F2 and entering `gnome-terminal`. If that doesn't work, please refer the documentation of your particular GNU/Linux distribution. Now, run `python3` and ensure there are no errors.

You can see the version of Python on the screen by running:

```
$ python3 -V
Python 3.5.1
```

NOTE: \$ is the prompt of the shell. It will be different for you depending on the settings of the operating system on your computer, hence I will indicate the prompt by just the \$ symbol.

CAUTION: Output may be different on your computer, depending on the version of Python software installed on your computer.

The “Hello World” Program

From now on, we will assume that you have Python installed on your system.

Next, we will write our first Python program.

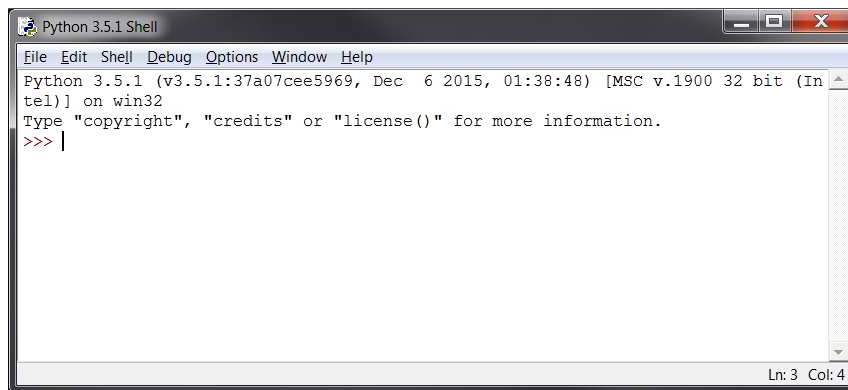
We will now see how to run a traditional 'Hello World' program in Python. This will teach you how to write, save and run Python programs.

There are two ways of using Python to run your program - using the **interactive interpreter prompt** or using a **source file**. We will now see how to use both of these methods.

Using the Interpreter Prompt

Open the terminal in your operating system (as discussed previously in the [Installation](#) chapter) or launch IDLE and use the Python shell. If you are on Windows or Raspbian, running the Python shell in IDLE is easiest.

Once you have started Python, you should see `>>>` where you can start typing stuff. This is called the *Python interpreter prompt*.

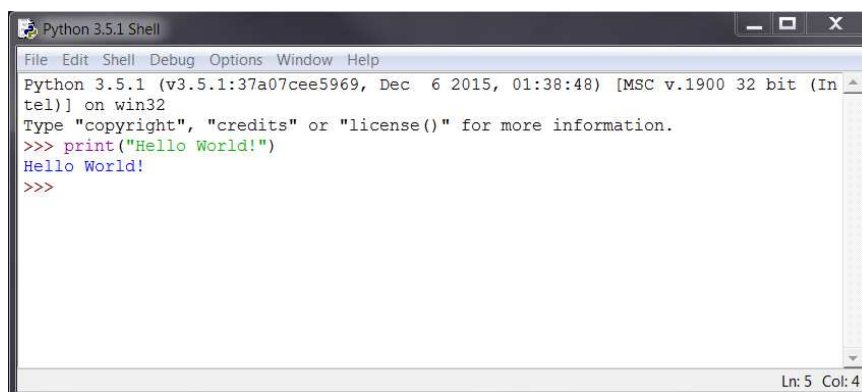


There is a tradition that whenever you learn a new programming language, the first program that you write and run is the 'Hello World' program - all it does is just say 'Hello World' when you run it. As Simon Cozens, the author of the amazing book *Beginning Perl*, says, it is the "traditional incantation to the programming gods to help you learn the language better."

At the Python interpreter prompt, type:

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

followed by the [enter] key. You should see the words `Hello World!` printed to the screen.



Notice that Python gives you the output of the line immediately! What you just entered is a single Python *statement*. We use `print` to (unsurprisingly) print any value that you supply to it. Here, we are supplying the text `Hello World!` and this is promptly printed to the screen.

Choosing an Editor

We cannot type out our program at the interpreter prompt every time we want to run something – it would take forever - so we have to save them in files. This way we can run our programs any number of times and make edits much quicker and easier.

To create our Python source files, we need an editing software where you can type and save. A good programmer's editor will make your life easier in writing the source files. Hence, the choice of an editor is crucial indeed. You have to choose an editor as you would choose a car you would buy. A good editor will help you write Python programs easily, making your journey more comfortable and helps you reach your destination (achieve your goal) in a much faster and safer way.

One of the very basic requirements is *syntax highlighting* where all the different parts of your Python program are colorized so that you can see your program and visualize its running.

If you are using Windows, *do not use Notepad* - it is a bad choice because it does not do syntax highlighting and also importantly it does not support indentation of the text which is very important in our case as we will see later. Good editors will automatically do this.

A few suggestions:

- IDLE – comes bundled with Python. Contains syntax highlighting. Easy to run programs. Probably your best choice at this level for Raspbian and Windows use.
- Notepad++ - a free text editor for Windows
- [PyCharm Educational Edition](#) - available on Windows, Mac OS X and GNU/Linux.
- [Vim](#) or [Emacs](#) - two of the most powerful editors and you will benefit from using them to write your Python programs.

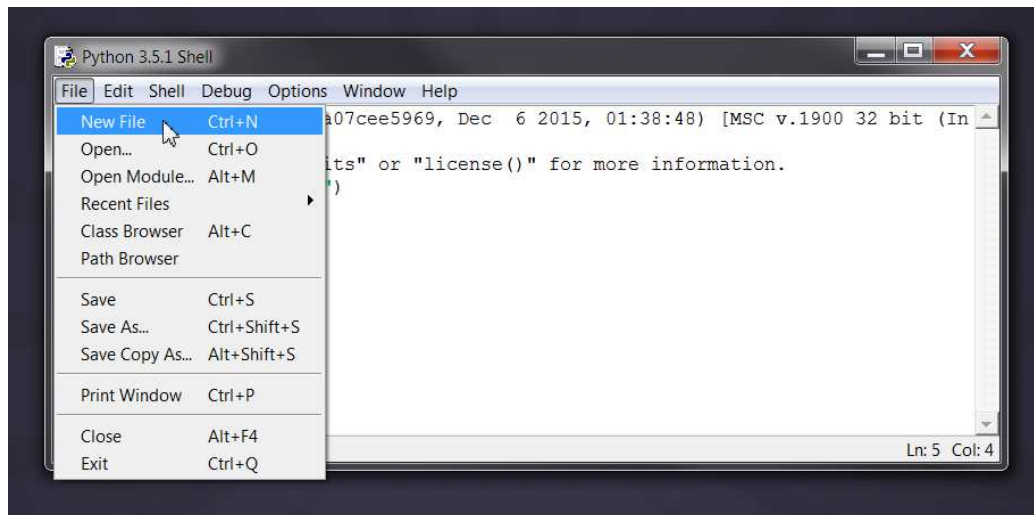
In case you are willing to take the time to learn Vim or Emacs, then I highly recommend that you do learn to use either of them as it will be very useful for you in the long run. However, as I mentioned before, beginners can start with IDLE, Notepad++, or PyCharm and focus the learning on Python rather than the editor at this moment.

Since IDLE comes bundles with Python, I will be using it in all future screen captures and directions.

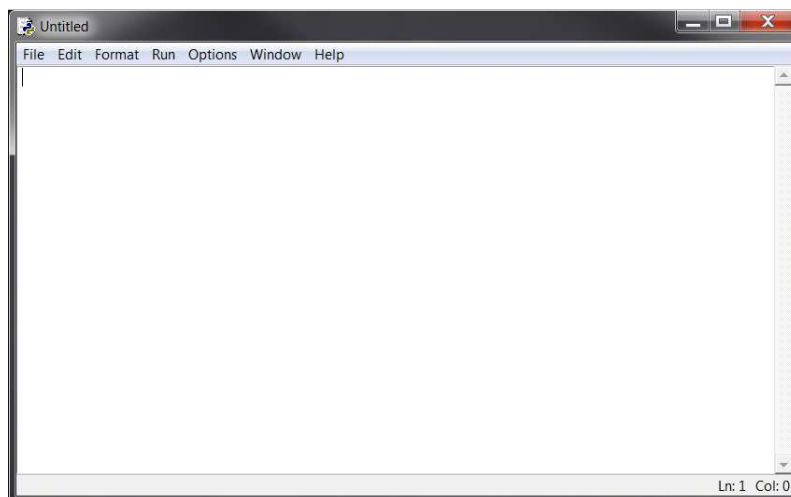
To reiterate, please **choose a proper editor** - it can make writing Python programs more fun and easy.

“Hello World!” Using a Source File

Now let's get back to programming. Start IDLE and open a new file by going to File > New File.



A lovely blank file will display:



The very first thing you need to do is **save this file** and give it a location and name that you will remember. For this exercise, we are going to be name it `hello.py`.

Go to File > Save As.

Find a good location for the file. Where should you save the file? To any folder for which you know the location of the folder. If you don't understand what that means, create a new folder and use that location to save and run all your Python programs.

Give it a name of `hello.py` and click Save. **IMPORTANT:** Always ensure that you give it the file extension of `.py`, for example, `foo.py`. IDLE does not automatically add the extension of `.py`. You need to do this yourself! If you forget, you will not see the fun syntax highlighting which makes editors so useful.

Now that we have opened a new file, saved it with the name "hello.py" we are now ready to write a program.

Type this:

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

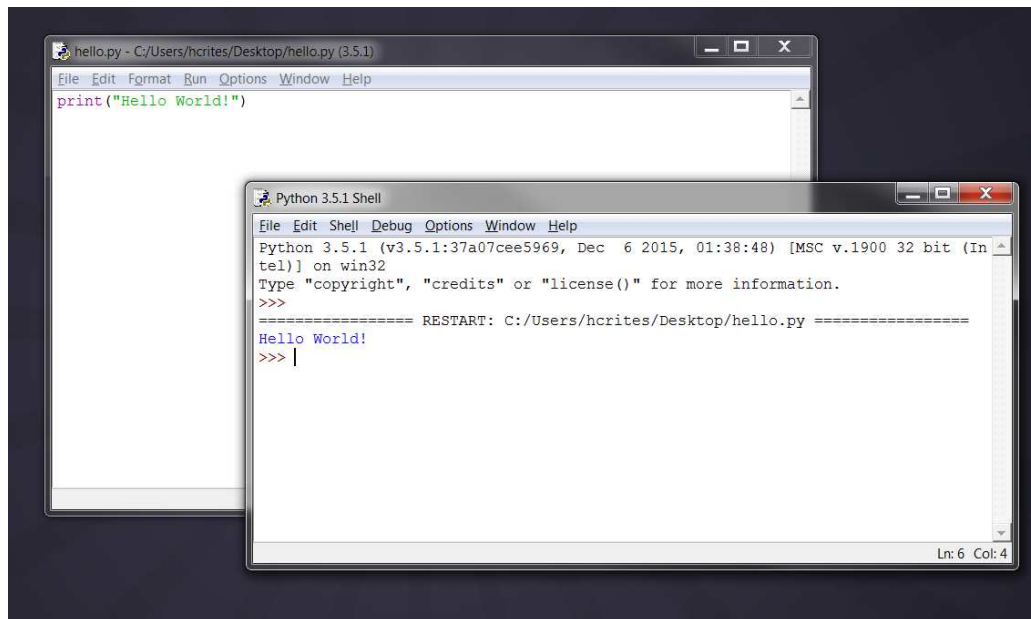
Save your file.

Now we are going to run the program. There's a few ways to do this. You can...

1. Open a terminal window, find the file, and execute it.

2. Double click the program to launch the terminal window
3. Run it in IDLE.

Since we are already using IDLE, it makes sense to run it using IDLE. To do this, go to Run > Run Module or in Windows, hit F5. The Python Shell should display and your program should execute:



If you got the output as shown above, congratulations! - you have successfully run your first Python program. You have successfully crossed the hardest part of learning programming, which is, getting started with your first program!

In case you got an error, please type the above program *exactly* as shown above and run the program again. Note that Python is case-sensitive i.e. `print` is not the same as `Print` - note the lowercase `p` in the former and the uppercase `P` in the latter. Also, ensure there are no spaces or tabs before the first character in each line - we will see why this is important later.

How “Hello World!” Works

Code	Output	Explanation
<code>print("Hello World!")</code>	Hello World!	A Python program is composed of <i>statements</i> . In our first program, we have only one statement. In this statement, we call the <code>print</code> <i>statement</i> to which we supply the text "Hello world!".

You should now be able to write, save and run Python programs at ease.



Go online to complete the
Hello World! Challenge

Formal and Natural languages

Natural languages are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express computations.

Formal languages tend to have strict syntax rules that govern the structure of statements. For example, in mathematics the statement $3 + 3 = 6$ has correct syntax, but $3 + = 3 \$ 6$ does not. In chemistry H_2O is a syntactically correct formula, but $_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with $3 += 3 \$ 6$ is that $\$$ is not a legal token in mathematics (at least as far as I know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz .

The second type of syntax rule pertains to the way tokens are combined. The equation $3 += 3$ is illegal because even though $+$ and $=$ are legal tokens, you can't have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

This is @ well-structured Engli\$h sentence with invalid t*kens in it.

This sentence all valid tokens has, but invalid structure with.

When you read a sentence in English or a statement in a formal language, you have to figure out the structure (although in a natural language you do this subconsciously). This process is called parsing.

Although formal and natural languages have many features in common—tokens, structure, and syntax—there are some differences:

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. If I say, “The penny dropped”, there is probably no penny and nothing dropping (this idiom means that someone understood something after a period of confusion). Formal languages mean exactly what they say.

Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. The difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

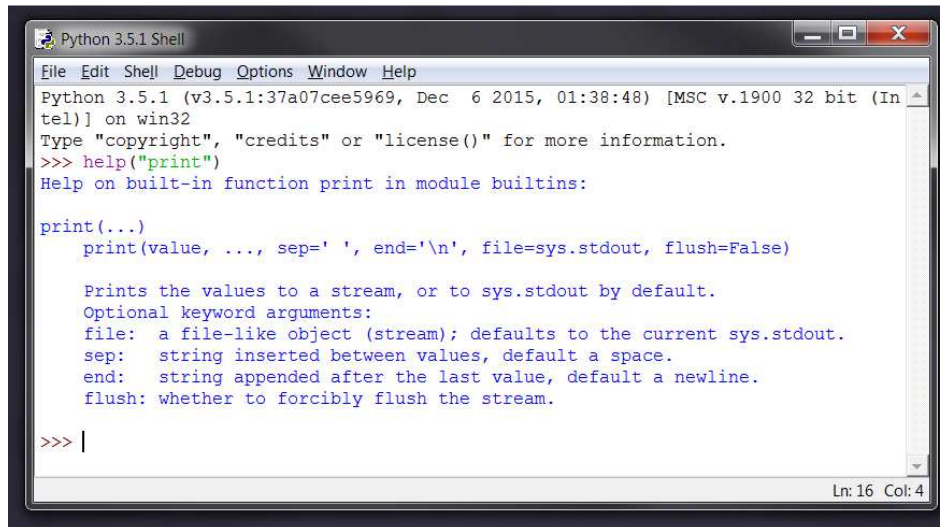
Prose: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Formal languages are denser than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

Getting Help

If you need quick information about any function or statement in Python, then you can use the built-in `help` functionality. This is very useful especially when using the interpreter prompt. For example, run `help('print')` - this displays the help for the `print` function which is used to display output onto the screen.

A screenshot of a Python 3.5.1 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following content:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> help("print")
Help on built-in function print in module builtins:

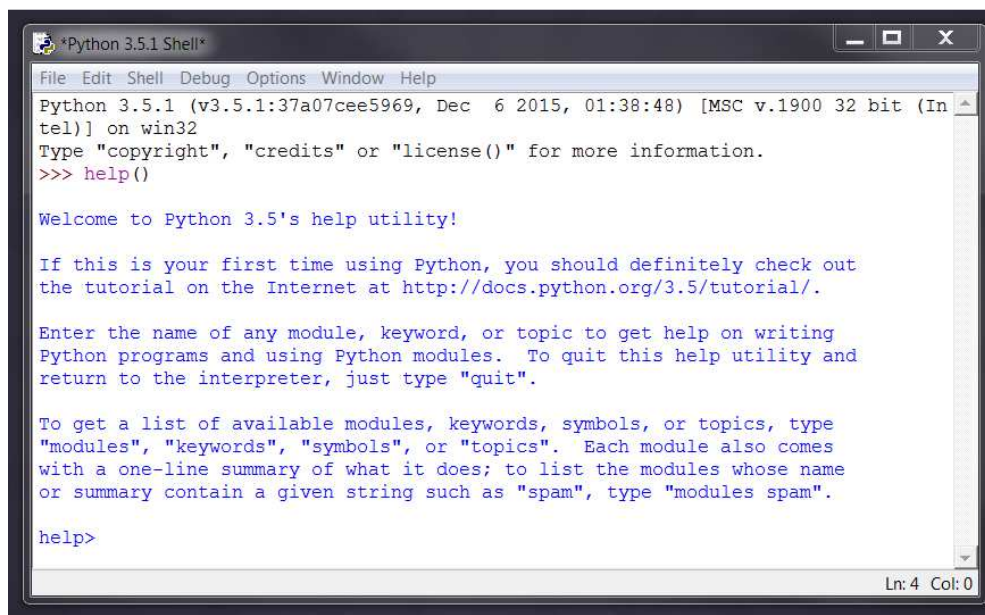
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep: string inserted between values, default a space.
    end: string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.

>>> |
```

The status bar at the bottom right indicates 'Ln: 16 Col: 4'.

Similarly, you can obtain information about almost anything in Python. Use `help()` to learn more about using `help` itself!

A screenshot of a Python 3.5.1 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following content:

```
*Python 3.5.1 Shell*
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> help()

Welcome to Python 3.5's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.5/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

The status bar at the bottom right indicates 'Ln: 4 Col: 0'.

In case you need to get help for operators like `return`, then you need to put those inside quotes such as `help('return')` so that Python doesn't get confused on what we're trying to do.

Debugging, Part 1

Programmers make mistakes. For whimsical reasons, programming errors are called bugs and the process of tracking them down is called debugging.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed.

There is evidence that people naturally respond to computers as if they were people. When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture.

Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each lesson there is a section, like this one, with suggestions for debugging. I hope they help!

Practice Makes Perfect

It is a good idea to read this book in front of a computer so you can try out the examples as you go. The online version of this book will have the exercises embedded in the correct location. However, if decide to read the printed version, please stop and take the time to try out the exercise before moving on to the next section.

Exercise 1

Whenever you are experimenting with a new feature, you should **try to make mistakes**.

For example, in the “Hello, world!” program,

- What happens if you leave out one of the quotation marks?
- What if you leave out both?
- What if you spell print wrong?

This kind of experiment helps you remember what you read; it also helps when you are programming, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later and accidentally.

1. In a print statement, what happens if you leave out one of the parentheses, or both?
2. If you are trying to print a string, what happens if you leave out one of the quotation marks, or both?
3. You can use a minus sign to make a negative number like -2. What happens if you put a plus sign before a number? What about 2++2?
4. In math notation, leading zeros are ok, as in 02. What happens if you try this in Python?
5. What happens if you have two values with no operator between them?

Basics

Just printing `Hello World!` is not enough, is it? You want to do more than that - you want to take some input, manipulate it and get something out of it. We can achieve this in Python using constants and variables, and we'll learn some other concepts as well in this chapter.

Values and Types

A value is one of the basic things a program works with, like a letter or a number. Some values we have seen so far are `2`, `42.0`, and `'Hello, World!'`.

These values belong to different types: `2` is an integer, `42.0` is a floating-point number, and `'Hello, World!'` is a string, so-called because the letters it contains are strung together.

If you are not sure what type a value has, the interpreter can tell you:

```
>>> type(2)
<class 'int'>

>>> type(42.0)
<class 'float'>

>>> type('Hello, World!')
<class 'str'>
```

In these results, the word “class” is used in the sense of a category; a type is a category of values.

Not surprisingly, integers belong to the type `int`, strings belong to `str` and floating-point numbers belong to `float`.

Each of these values – `2`, `42.0`, and `“Hello World!”` are examples of **literal constants**. They are called a literal because it is *literal* - you use its value literally. The number `2` always represents itself and nothing else - it is a *constant* because its value cannot be changed. Hence, all these are referred to as literal constants.

Numbers

Numbers are mainly of two types - **integers** and **floating-point numbers**.

An example of an integer is `2` which is just a whole number. Other integers are `-5`, `0`, and `124789`.

Floating-point numbers (or *floats* for short) are numbers with a decimal. Some examples include `3.23` and `52.3E-4`. The `E` notation indicates powers of 10. In this case, `52.3E-4` means `52.3 * 10^-4` or `0.000523`.

Note for Experienced Programmers: *There is no separate `long` type. The `int` type can be an integer of any size.*

When you type a large integer, you might be tempted to use commas between groups of digits, as in `1,000,000`. This is not a legal *integer* in Python, but it is legal syntax:

Code	Output
<code>1,000,000</code>	<code>(1, 0, 0)</code>

That's not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers. We'll learn more about this kind of sequence later.

Strings

A string is a *sequence of characters*. Strings are basically just a bunch of words. In our Hello World! program, "Hello World!" was a string. We used the `print` statement to display the string.

You will be using strings in almost every Python program that you write, so pay attention to the following part.

Single Quote

You can specify strings using single quotes such as `'Quote me on this'`.

All white space i.e. spaces and tabs, within the quotes, are preserved as-is.

Code	Output
<code>print('Quote me on this')</code>	Quote me on this



Go online to complete the
Single Quotes Challenge

Double Quotes

Strings in double quotes work exactly the same way as strings in single quotes. An example is `"What's your name?"`.

Code	Output
<code>print("What's your name?")</code>	What's your name?



Go online to complete the
Double Quotes Challenge

Triple Quotes

You can specify multi-line strings using triple quotes - (`"""` or `'''`). You can use single quotes and double quotes freely within the triple quotes. An example is:

Code	Output
<pre>print(''This is a multi-line string. This is the second line. "What's your name?," I asked. He said "Bond, James Bond." ''')</pre>	<pre>This is a multi-line string. This is the second line. "What's your name?," I asked. He said "Bond, James Bond."</pre>



Go online to complete the
Triple Quotes Challenge

Strings Are Immutable

This means that once you have created a string, you cannot change it. Although this might seem like a bad thing, it really isn't. We will see why this is not a limitation in the various programs that we see later on.

Note for C/C++ Programmers: *There is no separate `char` data type in Python. There is no real need for it and I am sure you won't miss it.*

Note for Perl/PHP Programmers: *Remember that single-quoted strings and double-quoted strings are the same - they do not differ in any way.*

What about values like `'2'` and `'42.0'`? They look like numbers, but they are in quotation marks like strings.

```
>>> type('2')
<class 'str'>

>>> type('42.0')
<class 'str'>
```

They're strings.

Escape Sequences

Suppose, you want to have a string which contains a single quote (`'`), how will you specify this string? For example, the string is `"What's your name?"`. You cannot specify `'What's your name?'` because Python will be confused as to where the string starts and ends. So, you will have to specify that this single quote does not indicate the end of the string.

This can be done with the help of what is called an *escape sequence*. You specify the single quote as `\'` : notice the backslash. Now, you can specify the string as `'What\'s your name?'`.

Code	Output
<code>print('What\'s your name?')</code>	What's your name?

Another way of specifying this specific string would be "What's your name?" i.e. using double quotes. Similarly, you have to use an escape sequence for using a double quote itself in a double quoted string. Also, you have to indicate the backslash itself using the escape sequence `\\`.

Code	Output
<code>print("He said, \"Hi!\"")</code>	He said, "Hi!"
<code>print("This is a backslash: \\")</code>	This is a backslash: \



Go online to complete the
Using Escape Characters Challenge

What if you wanted to specify a two-line string? One way is to use a triple-quoted string as shown [previously](#) or you can use an escape sequence for the newline character - `\n` to indicate the start of a new line. An example is:

Code	Output
<code>print('Line 1\nLine 2')</code>	Line 1 Line 2

Another useful escape sequence to know is the tab: `\t`. This will insert a tab into your string which can be useful for formatting:

Code	Output
<code>print('Column 1\tColumn 2')</code>	Column 1 Column 2

There are many more escape sequences but I have mentioned only the most useful ones here.

One thing to note is that in a string, a single backslash at the end of the line indicates that the string is continued in the next line, but no newline is added. For example:

```
"Sentence 1. \
Sentence 2."
```

is equivalent to:

```
"Sentence 1. Sentence 2"
```

Code	Output
------	--------

```
print("Sentence 1. \
Sentence 2.")
```

Sentence 1. Sentence 2.



Go online to complete the
Special Escape Sequences Challenge

Raw String

If you need to specify some strings where no special processing such as escape sequences are handled, then what you need is to specify a *raw* string by prefixing `r` or `R` to the string. An example is:

```
r"Newlines are indicated by \n"
```

Code	Output
<code>print(r"Newlines are indicated by \n")</code>	Newlines are indicated by \n

See how the `\n` is not rendered as a new line, but is displayed as-is.

Note for Regular Expression Users: Always use raw strings when dealing with regular expressions. Otherwise, a lot of backwhacking may be required. For example, backreferences can be referred to as `'\1'` or `r'\1'`.



Go online to complete the
Raw Strings Challenge

End Behavior

The `print` statement automatically ends each statement with a line break.

Code	Output
<code>print("Line 1")</code> <code>print("Line 2")</code>	Line 1 Line 2
<code>print("Line 1\nLine 2")</code> <code>print("Line 3")</code>	Line 1 Line 2 Line 3

Later we'll learn how to control the end behavior of the `print` statement. For now, recognize that there will always be a line break at the end of each `print` statement.

Variables

Using just literal constants can soon become boring and impractical - we need some way of storing any information and manipulate them as well. This is where *variables* come into the picture. Variables are exactly what the name implies - their value can vary, i.e., you can store anything using a variable. Variables are just parts of your computer's memory where you store some information. Unlike literal constants, you need some method of accessing these variables and hence you give them names.

Identifier Naming

Variables are examples of identifiers. *Identifiers* are names given to identify *something*. There are some rules you have to follow for naming identifiers:

- The first character of the identifier must be a letter of the alphabet (uppercase ASCII or lowercase ASCII or Unicode character) or an underscore (_).
- The rest of the identifier name can consist of letters (uppercase ASCII or lowercase ASCII or Unicode character), underscores (_) or digits (0-9).
- Identifier names are case-sensitive. For example, `myname` and `myName` are *not* the same. Note the lowercase `n` in the former and the uppercase `N` in the latter.

Examples of *valid* identifier names are:

- `I`
- `name_2_3`
- `_part_1`

Examples of *invalid* identifier names are:

- `2things`
- `this is spaced out`
- `my-name`
- `>a1b2_c3`

If you give a variable an illegal name, you get a syntax error:

Code	Output
<code>76trombones = 'big parade'</code>	<code>SyntaxError: invalid syntax</code>
<code>more@ = 1000000</code>	<code>SyntaxError: invalid syntax</code>
<code>class = 'Advanced Theoretical Zymurgy'</code>	<code>SyntaxError: invalid syntax</code>

`76trombones` is illegal because it begins with a number. `more@` is illegal because it contains an illegal character, `@`. But what's wrong with `class`?

It turns out that `class` is one of Python's keywords. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python 3 has these keywords:

<code>False</code>	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>
<code>None</code>	<code>continue</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
<code>True</code>	<code>def</code>	<code>from</code>	<code>nonlocal</code>	<code>while</code>
<code>and</code>	<code>del</code>	<code>global</code>	<code>not</code>	<code>with</code>
<code>as</code>	<code>elif</code>	<code>if</code>	<code>or</code>	<code>yield</code>
<code>assert</code>	<code>else</code>	<code>import</code>	<code>pass</code>	
<code>break</code>	<code>except</code>	<code>in</code>	<code>raise</code>	

You don't have to memorize this list. In most development environments, keywords are displayed in a different color; if you try to use one as a variable name, you'll know.

Data Types

Variables can hold values of different types called data types. The basic types are numbers and strings, which we have already discussed. In later lessons, we will see how variables can hold other data types.

Assignment Statements

An assignment statement creates a new variable and gives it a value:

```
message = 'And now for something completely different'
n = 17
pi = 3.141592653589793
```

This example makes three assignments. The first assigns a string to a new variable named `message`; the second gives the integer 17 to `n`; the third assigns the (approximate) value of π to `pi`.

If we were to display the variables using the `print` statement, we would see their values:

Code	Output
<pre>message = 'And now for something completely different' print(message)</pre>	And now for something completely different
<pre>n = 17 print(n)</pre>	17
<pre>pi = 3.141592653589793 print(pi)</pre>	3.141592653589793



Go online to complete the Print a Variable Challenge

You can make more than one assignment to the same variable. In fact, in your programs, this will happen frequently. A new assignment will make the existing variable point to a new value – and stop referring to the old value. To make a new assignment, you simply write a new assignment statement.

Code	Output
<code>x = 5</code>	
<code>print(x)</code>	5
<code>x = 6</code>	
<code>print(x)</code>	6

Variables can also switch data types:

Code	Output
<code>my_variable = 5</code>	
<code>print(my_variable)</code>	5
<code>my_variable = "Good morning!"</code>	
<code>print(my_variable)</code>	Good morning!



Go online to complete the Print a Variable, Part 2 Challenge

Output with Variables and Literal Constants

There are times when we need to display one or more variables and literal constants together. The print statement has already got you covered. You simply separate the variables and literal constants using commas:

Code	Output
<code>my_name = "Heather"</code>	
<code>print("Hi,", my_name)</code>	Hi, Heather

Notice that I did not explicitly put a space after the comma – Python added the space itself. Later, we will learn how to control the character which separates components of the `print` statement.

Here are some more examples:

Code	Output
<pre>fav_color = 'blue' print("I like", fav_color, "too!")</pre>	I like blue too!
<pre>animal = "dogs" number = 5 print('I see', number, animal, "running")</pre>	I see 5 dogs running



Go online to complete the
Printing Variables and Strings Challenge



Go online to complete the
Variable Assignment Challenge



Go online to complete the
Variable Reassignment Challenge

Example: Using Variables and Literal Constants

Type and run the following program:

```
i = 5
print(i)
i = "hello"
print(i)
s = '''This is a multi-line string.
This is the second line.'''
print(s)
```

Output:

```
5
hello
This is a multi-line string.
This is the second line.
```

How It Works

Here's how this program works.

Code	Output	Explanation
i = 5	<i>none</i>	First, we assign the literal constant value 5 to the variable <code>i</code> using the assignment operator (<code>=</code>). This line is called a statement because it states that something should be done and in this case, we connect the variable name <code>i</code> to the value 5.
print(i)	5	Next, we print the value of <code>i</code> using the <code>print</code> statement which, unsurprisingly, just prints the value of the variable to the screen.
i = "hello"	<i>none</i>	Then we reassign <code>i</code> to the string "hello".
print(i)	6	We then print it and expectedly, we get the value 6 .
s = '''This is a multi-line string. This is the second line.'''	<i>none</i>	Similarly, we assign the literal string to the variable <code>s</code> .
print(s)	This is a multi-line string. This is the second line.	Then we print it.

Expressions, Operators, and Statements

Time for a little more lingo. All programs will contain a combination of expressions, operators, and statements.

Expressions

Most programs that you write will contain **expressions**. An expression is a combination of values, variables, and operators. A simple example of an expression is $2 + 3$. When you type an expression at the prompt, the interpreter evaluates it, which means that it finds the value of the expression. In this case, the expression $2 + 3$ evaluates to 5 (because $2 + 3 = 5$).

However, a value all by itself is also considered an expression, and so is a variable, so the following are all legal expressions:

Expression	Value	Notes
42	42	
n	17	
n + 25	42	In this example, n has the value 17 and n + 25 has the value 42 because $17 + 25 = 42$.

Operators

Operators are functionality that do something and can be represented by symbols such as + or by special keywords. Operators require some data to operate on and such data is called *operands*. In this case, 2 and 3 are the operands.

The following are expressions which contain an operator:

Expression	Value	Notes
2 + 3	5	We are performing addition in this example. The operator is the plus sign.
3 * 5	15	We are performing multiplication in this example. The operator is the multiplication sign.

Statements

A **statement** is a unit of code that has an effect, like creating a variable or displaying a value. Statements contain expressions. Expressions may contain one or more operators.

```
n = 17
print(n)
```

The first line is an assignment statement that gives a value to `n`. The second line is a `print` statement that displays the value of `n`.

When you type a statement, the interpreter executes it, which means that it does whatever the statement says. In general, statements don't have values – they perform actions.

Mathematical Operators

Here is a brief run through of the most common mathematical operators in Python. A special thing to note is that these operators can work on all data types – not just integers and floats. Strings, sequences, and objects can all use these mathematical operators.

+ (plus)

The plus `+` operator performs addition on two objects.

Code	Output	Notes
<code>3 + 5</code>	<code>8</code>	Notice that the output is an integer.
<code>1.5 + 2.5</code>	<code>4.0</code>	Notice that the output is a float.
<code>1 + 1.0</code>	<code>2.0</code>	Notice that we are adding an integer and a float. The output is a float.
<code>'a' + 'b'</code>	<code>'ab'</code>	



Go online to complete the
Calculations - Addition Challenge

- (minus)

The minus `-` operator gives the subtraction of one number from the other; if the first operand is absent it is assumed to be zero. Minus does not work with strings.

Code	Output
<code>-5.2</code>	<code>-5.2</code>
<code>50 - 24</code>	<code>26</code>



Go online to complete the
Calculations - Subtraction Challenge

* (multiply)

The multiply operator gives the multiplication of the two numbers or returns the string repeated that many times.

Code	Output
<code>2 * 3</code>	6
<code>'la' * 3</code>	'lalala'



Go online to complete the
Calculations - Multiplication Challenge

** (power)

The power operator performs an exponentiation; that is, it returns x to the power of y . Power does not work with strings.

Code	Output	Notes
<code>3 ** 4</code>	81	This is the same as 3^4 which is the same as $3 * 3 * 3 * 3$
<code>4 ** 2</code>	16	This is the same as 4^2 which is the same as $4 * 4 = 16$

/ (divide)

The division / operator divides x by y . Python will always return a float. Divide does not work with strings.

Code	Output	Notes
<code>13 / 3</code>	4.333333333333333	Note that this returns a float

<code>4 / 2</code>	<code>2.0</code>	Note that even though $4 / 2 = 2$, python returns a float and not an integer.
<code>0 / 5</code>	<code>0.0</code>	Note that even though $0 / 5 = 0$, python returns a float and not an integer.

// (divide and floor)

The floor division operator, `//`, divides two numbers and rounds down to an integer. Divide and floor does not work with strings.

Code	Output	Notes
<code>13 // 3</code>	<code>4</code>	Note that this returns an integer
<code>4 // 2</code>	<code>2</code>	Note that this returns an integer
<code>0 // 5</code>	<code>0</code>	Note that this returns an integer
<code>-13 // 5</code>	<code>-3</code>	Note that this returns an integer

When would you want to use this? Suppose the run time of a movie is 105 minutes. You might want to know how long that is in hours. Conventional division returns a floating-point number:

```
>>> minutes = 105
>>> minutes / 60
1.75
```

But we don't normally write hours with decimal points. Floor division returns the integer number of hours, dropping the fraction part:

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

To get the remainder, you could subtract off one hour in minutes:

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```



Go online to complete the
Calculations - Division Challenge

% (modulo)

The modulus operator % divides two numbers and returns the remainder.

Code	Output	Notes
13 % 3	1	Note that this returns an integer
-25.5 % 2.25	1.5	Note that this returns a float

We can use the modulus operator % on our movie running time example.

```
>>> remainder = minutes % 60
>>> remainder
45
```

The modulus operator is more useful than it seems. For example, you can check whether one number is divisible by another—if $x \% y$ is zero, then x is divisible by y .

Also, you can extract the right-most digit or digits from a number. For example, $x \% 10$ yields the right-most digit of x (in base 10). Similarly $x \% 100$ yields the last two digits.



Go online to complete the
Calculations - Modulo Challenge



Go online to complete the
Calculations – Number Trick Challenge

Shortcut for Math Operation and Assignment

It is common to run a math operation on a variable and then assign the result of the operation back to the variable, hence there is a shortcut for such expressions:

```
a = 2
a = a * 3
```

can be written as:

```
a = 2
a *= 3
```

Notice that `variable = variable operation expression` becomes `variable operation= expression`.

Other examples:

Long Code

Shortcut Code

<code>a = a + 5</code>	<code>a += 5</code>
<code>a = a - 2.25</code>	<code>a -= 2.25</code>
<code>a = a * 4</code>	<code>a *= 4</code>
<code>a = a / 3</code>	<code>a /= 3</code>

Order of Operations

If you had an expression such as $2 + 3 * 4$, is the addition done first or the multiplication?

When an expression contains more than one operator, the order of evaluation depends on the order of operations. For mathematical operators, Python follows mathematical convention. The acronym PEMDAS is a useful way to remember the rules:

- **Parentheses** have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, $2 * (3-1)$ is 4, and $(1+1)**(5-2)$ is 8. You can also use parentheses to make an expression easier to read, as in $(\text{minute} * 100) / 60$, even if it doesn't change the result.
- **Exponentiation** has the next highest precedence, so $1 + 2**3$ is 9, not 27, and $2 * 3**2$ is 18, not 36.
- **Multiplication and Division** have higher precedence than Addition and Subtraction. So $2*3-1$ is 5, not 4, and $6+4/2$ is 8, not 5.
- **Operators** with the same precedence are evaluated from left to right (except exponentiation). So in the expression $\text{degrees} / 2 * \pi$, the division happens first and the result is multiplied by pi. To divide by 2π , you can use parentheses or write $\text{degrees} / 2 / \pi$.

I don't work very hard to remember the precedence of operators. If I can't tell by looking at the expression, I use parentheses to make it obvious.

For example, $2 + (3 * 4)$ is definitely easier to understand than $2 + 3 * 4$ which requires knowledge of the operator precedences. As with everything else, the parentheses should be used reasonably (do not overdo it) and should not be redundant, as in $(2 + (3 * 4))$.

There is an additional advantage to using parentheses - it helps us to change the order of evaluation. For example, if you want addition to be evaluated before multiplication in an expression, then you can write something like $(2 + 3) * 4$.

Operators are usually associated from left to right. This means that operators with the same precedence are evaluated in a left to right manner. For example, $2 + 3 + 4$ is evaluated as $(2 + 3) + 4$.



Go online to complete the
Calculations – Order of Operations Challenge

Example: Calculating on a Rectangle

Type and run the following program:

```
length = 5
breadth = 2
area = length * breadth
print('Area is', area)
print('Perimeter is', 2 * (length + breadth))
```

Output:

```
Area is 10
Perimeter is 14
```

How It Works

Here's how this program works.

Code	Output	Explanation
<code>length = 5</code>	<i>none</i>	First, we assign the literal constant value 5 to the variable <code>length</code> using the assignment operator (<code>=</code>).
<code>breadth = 2</code>	<i>none</i>	Next, we assign the literal constant value 2 to the variable <code>breadth</code> using the assignment operator (<code>=</code>).
<code>area = length * breadth</code>	<i>none</i>	We use these to calculate the area and perimeter of the rectangle with the help of expressions. We store the result of the expression <code>length * breadth</code> in the variable <code>area</code> .
<code>print('Area is', area)</code>	Area is 10	Next, we print the value of <code>area</code> using the <code>print</code> statement.
<code>print('Perimeter is', 2 * (length + breadth))</code>	Perimeter is 14	Finally, we directly use the value of the expression <code>2 * (length + breadth)</code> in the <code>print</code> function to display the perimeter of the rectangle.

Notice how Python *pretty-prints* the output. Even though we have not specified a space between `'Area is'` and the variable `area`, Python puts it for us so that we get a clean nice output and the program is much more readable this way (since we don't need to worry about spacing in the strings we use for output). This is an example of how Python makes life easy for the programmer.

Getting Input from the Keyboard

There will be situations where your program has to interact with the user. For example, you would want to take information from the user and then display some results back. We can achieve this using the `input()` function and `print` function respectively.

The built-in function, `input`, stops the program and waits for the user to type something. When the user presses Return or Enter, the program resumes and `input` returns what the user typed as a string.

Look at the following example:

```
my_name = input('What is your name?')
print(my_name)
```

When we execute this code, the interpreter will prompt us for our name and allow us to enter whatever we like.



Code	Output	Explanation
<pre>my_name = input('What is your name?')</pre>	What is your name?	When this code is executed, the program displays the input statement's prompt string, in this case What is your name? and waits for the user to provide some input. The program pauses until the Enter or Return key is pressed. Then the program assigns the input received from the user – in this case the string Heather – to the variable <code>my_name</code> and continues.
<pre>print('Hi,', my_name)</pre>	Heather	Now that the program has received input from the user, we display the string <code>Hi</code> , followed by the value of <code>my_name</code> – Heather – using the <code>print</code> statement.

Notice that the input received from the user – in this case the string `Heather` – is right up against the prompt `What is your name?` This is rather ugly. You can make it look nice by either adding a space at the end of the prompt:

```

examples.py - C:/Users/hcrites/AppData/Local/Programs/Python/Python35-32/examples.py (3.5.1)
File Edit Format Run Options Window Help
my_name = input("What is your name? ")
print('Hi, ', my_name)

Python 3.5.1 Shell
File Edit Shell Debug Options Window Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> RESTART: C:/Users/hcrites/AppData/Local/Programs/Python/Python35-32/examples.py
What is your name? Heather
Hi, Heather
>>> |

```

Or you can cause the input to be entered on a new line by using a special sequence `\n`. As you should recall, the sequence `\n` at the end of the prompt represents a newline and causes a line break.

```

examples.py - C:/Users/hcrites/AppData/Local/Programs/Python/Python35-32/examples.py (3.5.1)
File Edit Format Run Options Window Help
my_name = input("What is your name?\n")
print('Hi, ', my_name)

Python 3.5.1 Shell
File Edit Shell Debug Options Window Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> RESTART: C:/Users/hcrites/AppData/Local/Programs/Python/Python35-32/examples.py
What is your name?
Heather
Hi, Heather
>>> |

```



Go online to complete the Basic Input Challenge

Converting Input to Integers and Floats

Also notice that everything which is entered by the user has a type of string. Even numbers. If you need to use the user's input to perform a mathematical calculation, you will need to convert the string to either an integer or to a float before you perform the calculation. To do this, you use the statements `int()` and `float()`.

Code	Output	Explanation
<pre>my_age = input('What is your age?') my_age = int(my_age)</pre>	<i>none</i>	<p>Get the user's age. It is automatically a string, even if the user types a number.</p> <p>Convert the the <code>my_age</code> variable to an integer using <code>int()</code>.</p> <p>Now you can perform mathematical calculations on the number.</p>
<pre>cost = input('Enter the cost')</pre>	<i>none</i>	<p>Get the cost of an item. It is automatically a string, even if the user types a number.</p>

```
cost = float(cost)
```

Convert the `cost` variable to an integer using `float()`.

Now you can perform mathematical calculations on the number.

However, if the user types something other than a valid number, you will get an error if you use `int()` or `float()`.

```
>>> age = input('How old are you?')
How old are you?
too old

>>> int(age)
ValueError: invalid literal for int() with base 10
```

We will see how to handle this kind of error later. But this is something to keep in mind – users will unintentionally mess up your programs and it is important to plan for it.



Go online to complete the
Input with Integers Challenge



Go online to complete the
Input with Floats Challenge

Debugging, Part 2

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

Syntax error:

“Syntax” refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so `(1 + 2)` is legal, but `8)` is a syntax error.

If there is a syntax error anywhere in your program, Python displays an error message and quits, and you will not be able to run the program. During the first few weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

Runtime error:

The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.

In the previous section, you saw that a `ValueError` was thrown when we tried to convert the string `old` to an integer. This is an example of a runtime error.

Semantic error:

The third type of error is “semantic”, which means related to meaning. If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

Writing Programs

Now that we have an idea of the basic types of data and we have seen how they can be received from users as input, stored in variables, manipulated with operators and statements, and displayed back on screen, we can begin to write programs that actually do useful things. First, however, there are some basic elements to a program that are important to use and important to understand.

Program Header

One of the biggest problems I have in day-to-day web application development is wasting time trying to figure out what the heck this 10-year-old file is and what it does. All of this frustration can be avoided by simply documenting your source code, starting with a program header. Just list the file name, the program name, the original author, creation date, and purpose. If you want to get fancy, you can add rows for modifications and edits. You can copy and paste this header template into the top of all your Python programs. Just remember to fill in the blanks.

```
# FILE:
# NAME:
# AUTHOR:
# DATE:
# PURPOSE:
```

Example:

```
# FILE:      tipping.py
# NAME:      Tipping Calculator
# AUTHOR:    Heather Crites
# DATE:      3/5/2017
# PURPOSE:   Calculates tips and bill based upon user input
```

Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments, and they start with the # symbol:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60      # percentage of an hour
```

Everything from the # to the end of the line is ignored—it has no effect on the execution of the program.

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out what the code does; it is more useful to explain why.

This comment is redundant with the code and useless:

```
v = 5      # assign 5 to v
```

This comment contains useful information that is not in the code:

```
v = 5      # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff.

Use as many useful comments as you can in your program to:

- explain assumptions
- explain important decisions
- explain important details
- explain problems you're trying to solve
- explain problems you're trying to overcome in your program, etc

[Code tells you how, comments should tell you why.](#)

This is useful for readers of your program so that they can easily understand what the program is doing. Remember, that person can be yourself after six months!

Variables

Unless you like writing [unmaintainable code](#), I suggest that you use descriptive names for most of your variables. While naming a variable `x` is short and sweet, it is pretty meaningless when you are trying to determine whether `x` is the tip or the tax rate or the user's input. Better variable names would be `tax_rate` or `TaxRate` or `varTaxRate`. Find a style you like and stick with it.

Indentation

Whitespace is important in Python. Actually, *whitespace at the beginning of the line is important*. This is called *indentation*. Leading whitespace (spaces and tabs) at the beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements.

This means that statements which go together *must* have the same indentation. Each such set of statements is called a *block*. We will see examples of how blocks are important in later chapters.

One thing you should remember is that wrong indentation can give rise to errors. For example:

```
i = 5
# Error below! Notice a single space at the start of the line
 print('Value is', i)
print('I repeat, the value is', i)
```

When you run this, you get the following error:

```
File "whitespace.py", line 3
 print('Value is', i)
 ^
IndentationError: unexpected indent
```

Did you notice that there is a single space at the beginning of the second line? The error indicated by Python tells us that the syntax of the program is invalid i.e. the program was not properly written. What this means to you is that *you cannot arbitrarily start new blocks of statements* (except for the default main block which you have been using all along, of course). Cases where you can use new blocks will be detailed in later lessons such as the [control flow](#).

Use four spaces for indentation. This is the official Python language recommendation. Good editors will automatically do this for you. Make sure you use a consistent number of spaces for indentation, otherwise your program will not run or will have unexpected behavior.



Go online to complete the
Fix this Code Challenge

Logical and Physical Line

A physical line is what you *see* when you write the program. A logical line is what *Python sees* as a single statement. Python implicitly assumes that each *physical line* corresponds to a *logical line*.

An example of a logical line is a statement like `print('hello world')` - if this was on a line by itself (as you see it in an editor), then this also corresponds to a physical line.

Implicitly, Python encourages the use of a single statement per line which makes code more readable.

If you want to specify more than one logical line on a single physical line, then you have to explicitly specify this using a semicolon (;) which indicates the end of a logical line/statement. For example:

```
i = 5  
print(i)
```

is effectively same as

```
i = 5;  
print(i);
```

which is also same as

```
i = 5; print(i);
```

and same as

```
i = 5; print(i)
```

However, I *strongly recommend* that you stick to *writing a maximum of a single logical line on each single physical line*. The idea is that you should never use the semicolon. In fact, I have *never* used or even seen a semicolon in a Python program.

There is one kind of situation where this concept is really useful: if you have a long line of code, you can break it into multiple physical lines by using the backslash. This is referred to as *explicit line joining*:

Code	Output
<pre>s = 'This is a string. \ This continues the string.' print(s)</pre>	<pre>This is a string. This continues the string.</pre>

Similarly,

```
i = \
5
```

is the same as

```
i = 5
```

Sometimes, there is an implicit assumption where you don't need to use a backslash. This is the case where the logical line has a starting parentheses, starting square brackets or a starting curly braces but not an ending one. This is called *implicit line joining*. You can see this in action when we write programs using lists in later lessons.

The Programming Process

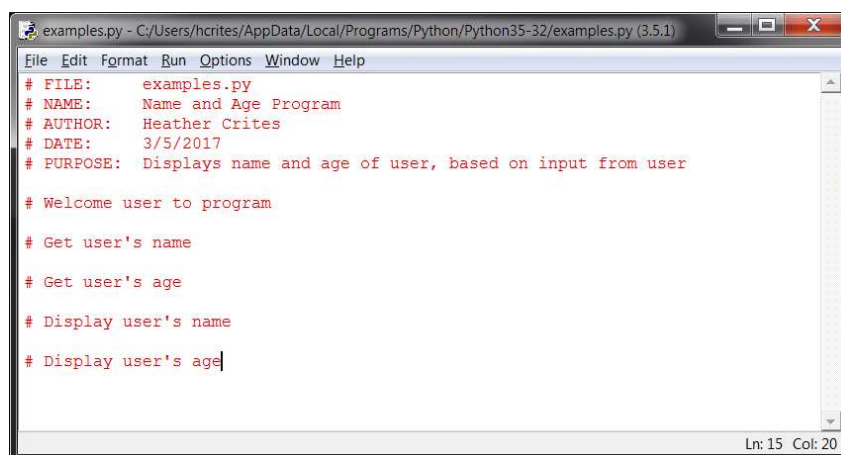
One of the most difficult hurdles when you are just getting started is how to start a new program or tackle an exercise or lab assignment from the beginning. First, recognize that very few programmers immediately start to write code when they build their programs. They create them in stages. You should to. Here are some simple steps to try:

1. **Use pseudocode to design the program:** sketch it out or write plain English to describe what you want the program to do. It is often most efficient to do this using comments within the source file itself.
2. **Fill in the pseudocode with real code:** start using your programming tools to fill in the blanks.
3. **Fix any Syntax errors:** You learned that syntax errors cause a program to not run at all. Try to run your program, then fix all of these.
4. **Test the program repeatedly:** use different inputs – negative numbers, letters, floats, and integers. See what sticks and what doesn't work. Fix some of your code if you don't get expected results.

Example:

You need to create a program which can accept a user's name and age, and displays it on the screen.

Step 1: Use pseudocode to design the program



```
examples.py - C:/Users/hcrites/AppData/Local/Programs/Python/Python35-32/examples.py (3.5.1)
File Edit Format Run Options Window Help
# FILE:      examples.py
# NAME:      Name and Age Program
# AUTHOR:    Heather Crites
# DATE:      3/5/2017
# PURPOSE:   Displays name and age of user, based on input from user

# Welcome user to program

# Get user's name

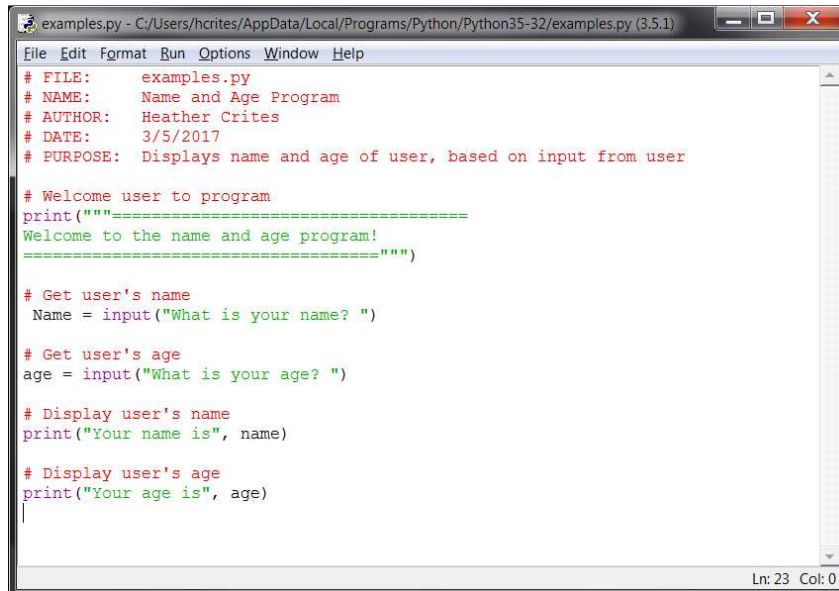
# Get user's age

# Display user's name

# Display user's age
```

In this example, I have written my header, then begin to sketch out the program using comments as pseudocode. I kept it simple and in plain terms. I don't need to write a novel.

Step 2: Fill in the pseudocode with real code



```
examples.py - C:/Users/hcrites/AppData/Local/Programs/Python/Python35-32/examples.py (3.5.1)
File Edit Format Run Options Window Help
# FILE:     examples.py
# NAME:     Name and Age Program
# AUTHOR:   Heather Crites
# DATE:    3/5/2017
# PURPOSE:  Displays name and age of user, based on input from user

# Welcome user to program
print("""=====
Welcome to the name and age program!
===== """)

# Get user's name
Name = input("What is your name? ")

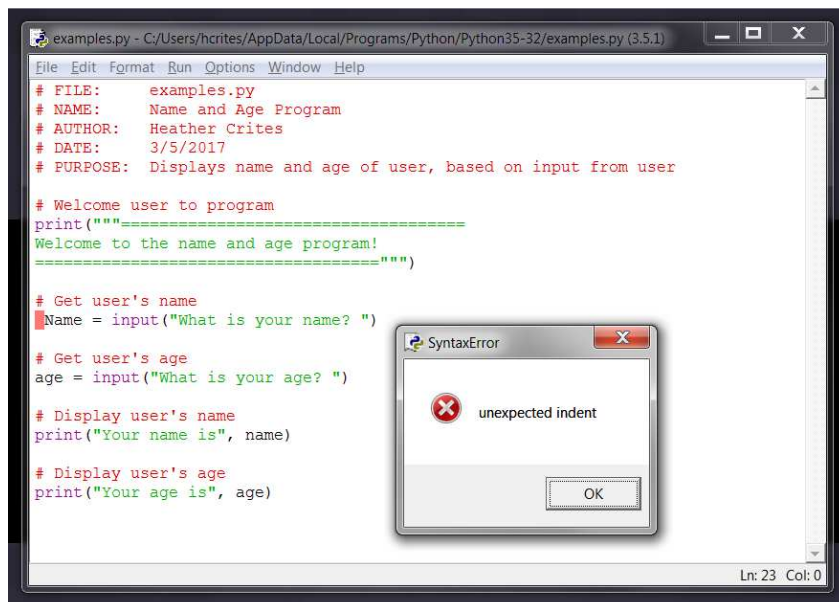
# Get user's age
age = input("What is your age? ")

# Display user's name
print("Your name is", name)

# Display user's age
print("Your age is", age)
```

Now I have filled in the blanks with code. For welcoming the user to the program, I used a `print` statement with a triple quote string. I created two variables to store the keyboard input received from the user for both the name and the age. I then returned the input back to the user with some extra remarks using the `print` statement.

Step 3: Fix any Syntax errors



```
examples.py - C:/Users/hcrites/AppData/Local/Programs/Python/Python35-32/examples.py (3.5.1)
File Edit Format Run Options Window Help
# FILE:     examples.py
# NAME:     Name and Age Program
# AUTHOR:   Heather Crites
# DATE:    3/5/2017
# PURPOSE:  Displays name and age of user, based on input from user

# Welcome user to program
print("""=====
Welcome to the name and age program!
===== """)

# Get user's name
Name = input("What is your name? ")

# Get user's age
age = input("What is your age? ")

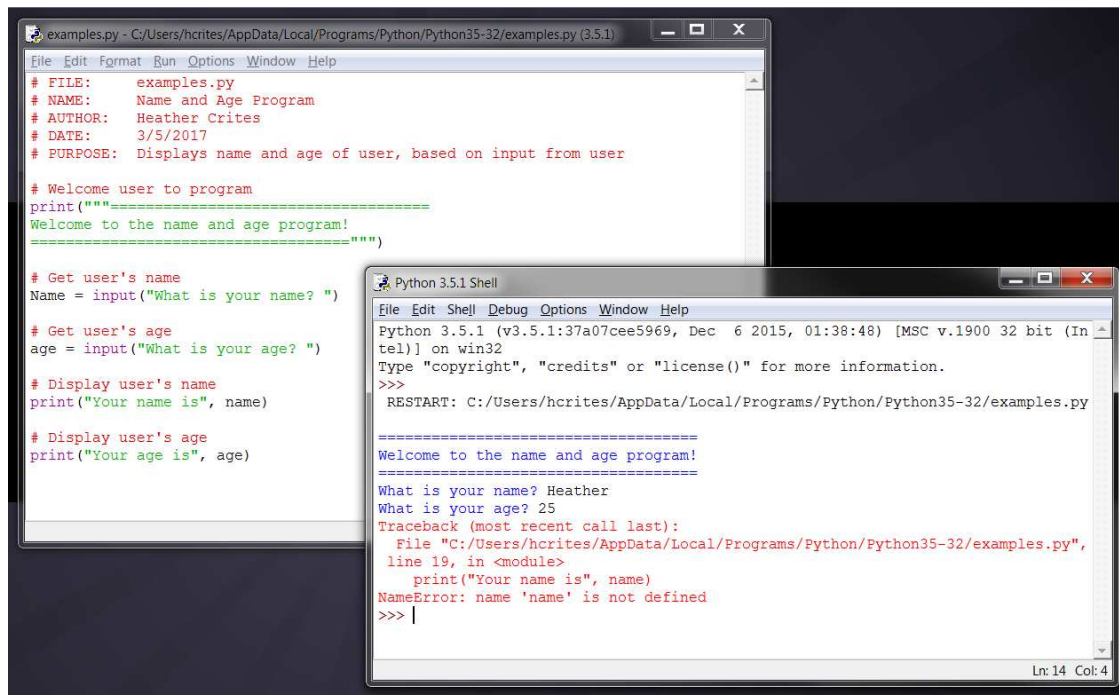
# Display user's name
print("Your name is", name)

# Display user's age
print("Your age is", age)
```

SyntaxError
unexpected indent
OK

Uh oh! I made a syntax mistake. It says an unexpected indent. This means I have an indentation problem. IDLE has highlighted a space in front of my variable called `Name`. This space shouldn't be here. I delete the space, correcting my indentation, and now I can run my program without a syntax error.

Step 4: Test



The screenshot shows a Python IDE window titled 'examples.py - C:/Users/hcrites/AppData/Local/Programs/Python/Python35-32/examples.py (3.5.1)'. The code in the editor is as follows:

```
# FILE:      examples.py
# NAME:      Name and Age Program
# AUTHOR:    Heather Crites
# DATE:      3/5/2017
# PURPOSE:   Displays name and age of user, based on input from user

# Welcome user to program
print("""=====
Welcome to the name and age program!
===== """)

# Get user's name
Name = input("What is your name? ")

# Get user's age
age = input("What is your age? ")

# Display user's name
print("Your name is", name)

# Display user's age
print("Your age is", age)
```

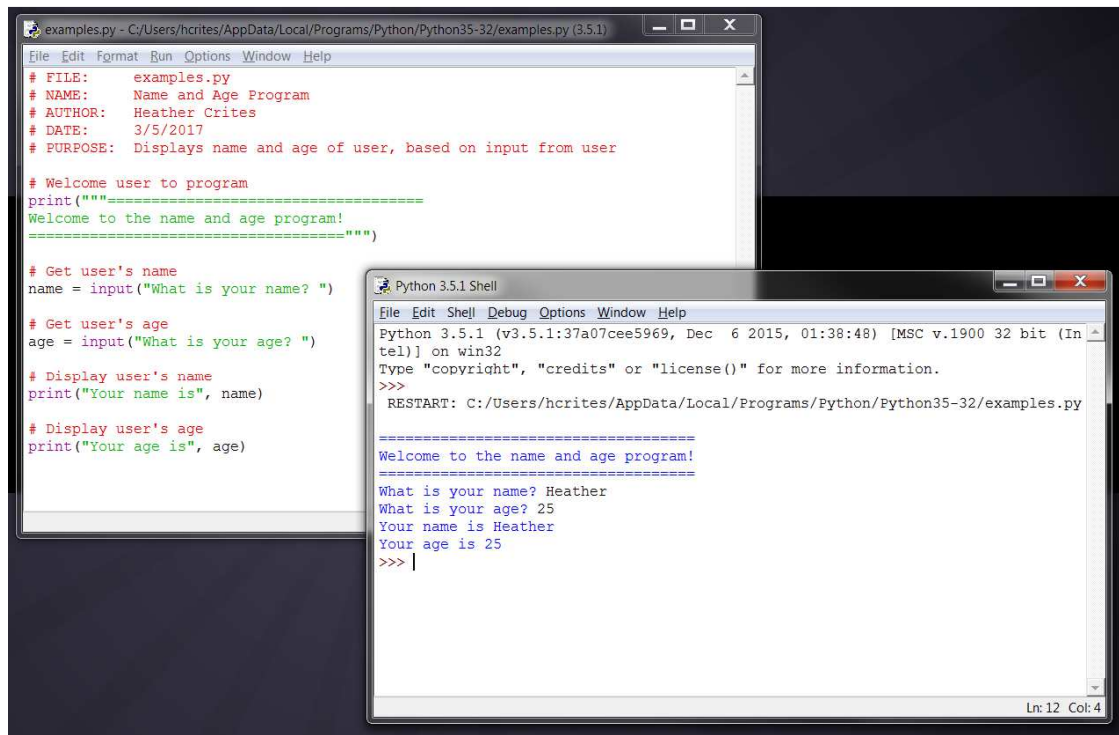
The Python 3.5.1 Shell window shows the following output:

```
Python 3.5.1 Shell
File Edit Shell Debug Options Window Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/hcrites/AppData/Local/Programs/Python/Python35-32/examples.py

=====
Welcome to the name and age program!
=====
What is your name? Heather
What is your age? 25
Traceback (most recent call last):
  File "C:/Users/hcrites/AppData/Local/Programs/Python/Python35-32/examples.py",
    line 19, in <module>
        print("Your name is", name)
NameError: name 'name' is not defined
>>> |
```

The error message indicates that the variable 'name' is not defined, despite being assigned to 'Name' in the code.

Now that I fixed my syntax error, my program runs. I put Heather in when prompted for the name and 25 when prompted for the age. Unfortunately, my program stopped running because of a runtime error. It says `NameError: name 'name' is not defined`. I look at my code and I see my problem. Variable names are case sensitive. I assigned the input received for the user's name to a variable called `Name` with an uppercase `N`, but then tried to display a variable called `name` with a lowercase `n`. These are two different variables and I never assigned any value to the lowercase version, which is why the error was thrown. I change `Name` to `name` and run it again:



The screenshot shows the same Python IDE window with the code from the previous screenshot, but with the variable `Name` changed to `name`:

```
# FILE:      examples.py
# NAME:      Name and Age Program
# AUTHOR:    Heather Crites
# DATE:      3/5/2017
# PURPOSE:   Displays name and age of user, based on input from user

# Welcome user to program
print("""=====
Welcome to the name and age program!
===== """)

# Get user's name
name = input("What is your name? ")

# Get user's age
age = input("What is your age? ")

# Display user's name
print("Your name is", name)

# Display user's age
print("Your age is", age)
```

The Python 3.5.1 Shell window shows the following output:

```
Python 3.5.1 Shell
File Edit Shell Debug Options Window Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/hcrites/AppData/Local/Programs/Python/Python35-32/examples.py

=====
Welcome to the name and age program!
=====
What is your name? Heather
What is your age? 25
Your name is Heather
Your age is 25
>>> |
```

The program now runs successfully and displays the user's name and age.

The whole program ran without errors! I run it a few more times and test different values until I am satisfied that it runs the way it is supposed to.

This is a very simple example, but it shows the common steps used.



Go online to complete the
Calculating the Area of a Triangle Challenge



Go online to complete the
Calculating Distance Challenge

Putting it All Together

1. Use a program header
2. Use comments
3. Name your variables well
4. Use appropriate indentation (4 spaces is best)
5. Understand the difference between a physical line and a logical line
6. Create your programs in stages