1 Input and Output

"The innermost of the four Galilean moons of Jupiter. It is named after ... one of the many lovers of Zeus (who is also known as Jupiter in the Roman mythology)." - Wikipedia, "Io"

In which we rescue the spaceship Discovery and learn the basics of chatting with Python.

Most of you have probably seen or read 2001, which means that I'll have to include a HAL quote in here eventually. But most of you probably *haven't* seen or read 2010, which is likely a function of it being not nearly as good, in my humble opinion. The next time you really feel like being depressed by strange science fiction writing, read "The Star" or "The Nine Billion Names of God." Make sure to do it after you've finished your homework, though.

Which, to get back on track, will likely have some sort of input and produce some sort of output. There are arguably plenty of interesting computer programs that accept no input; even in terms of mathematical functions, you could probably convince me that e or π are functions of a sort that accept no specific inputs and produce tremendously interesting outputs (if you're into those "number" things). Even constant functions like these are useful sometimes; a picture is in some sense a constant function that produces an array of red, green, and blue values as outputs. But that being said, you'll have a hard time finding an interesting computer program that doesn't produce any output!

We'll be a little more specific in our definitions of input and output here, though. In the real world, both input and output (often abbreviated I/O or just IO) can take many forms; a program might need a value or a whole bunch of values (e.g. from a microarray) to combine or calculate with, and its output might not be visible (e.g. it might change some files or silently dump a new value into memory someplace). In our discussion here, we'll be concentrating on two specific sources of input and output, traditionally called standard input and standard output.

1.1 Input

"Oregon. Home of the beaver. Famous for cherries."
"Beavers! Cherries! Input."
"I'm giving you great input!"
"More input. More input!" - Number 5 and Stephanie, "Short Circuit"

Standard input, or stdin as it's sometimes abbreviated, was originally "standard" because it was the only input. Back in the day (think 1960s), a program expected to be pretty much the only thing running on its host computer. It expected that computer to have one terminal, which would be a little text screen like your command prompt, and one punched tape or card reader. Once a program was loaded from tape/cards and was plugging along, the only place it could get input from was as a sequence of text data produced by a user typing something in on the console.

And that's exactly where standard input comes from still! Just as `print` displays text (in the form of strings) to your console, Python contains something named `sys.stdin` that will read text in from the console. You can access this input stream by importing the `sys` module. When you do, your program will stop in its tracks, wait for you to type something on the console (and press Enter), and return whatever you typed to your program as a string. This is not unlike what happens when you provide command line arguments; it just happens interactively instead of only when you start your program. `sys.stdout`, on the other hand, has functions like `write` that accept one string argument, display it on the terminal, and return nothing (very much like `print`, but without the trailing newline). `sys.stdin` acts like a collection (i.e. list) of input strings, one per line of text, and it also has functions like `readline` that accept no arguments, read in text from the terminal, and return one string; in a sense, input is purely symmetric with output. `sys.stdout.write` prints text to stdout (sans newline); `sys.stdin.readline` reads a line of text from stdin (including newline).

A Key Point

There's a magical key combination that can sometimes be useful for indicating the end of your input, traditionally referred to as the end of file or EOF sequence. Recall that pressing Control-C will immediately terminate your program. That's sometimes good, but often bad. Pressing Control-Z (on Windows) or Control-D (on a Macintosh) will *not* kill your program. What it does signal, though, is that you're done typing things on standard input. So if you're writing a program that, say, reads strings from stdin until it runs out of input, Control-Z or Control-D is

how you send it the no-more-stuff signal without terminating the whole program. Type in lines of text, one after the other, pressing Enter as necessary, and then press Control-Z plus Enter (Windows) or just Control-D (Mac) at the end. I don't think you'll necessarily need to use this any time soon, but it's important to know.

Eclipse Users, Read This!

Sometimes, things work differently in Eclipse than they do in a traditional environment. And sometimes they don't. Here you luck out - console input (and, as you've seen if you've tried Eclipse, output) is exactly the same in Eclipse as it is on the command line. To type input into your Python program, make sure your cursor is in the Console window and, well, type it in! It'll appear in snazzy cyan text and (other than the psychedelic coloration) behave exactly as it would in a command prompt window.

StdIn Quick Reference

Method	Type	Description
`sys.stdin.readline( )`	string	Read one string (up to a newline character) from console
`for strLine in sys.stdin:` `  *process strLine*`	-	Loop over lines (up to a newline character) from console

1.2 Output

> *"On two occasions, I have been asked [by members of Parliament], 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able to rightly apprehend the kind of confusion of ideas that could provoke such a question." - Charles Babbage, Harper's "New Monthly Magazine" 1864*

You should already have a pretty good idea of how output behaves in Python, since you've been producing all kinds of crazy stuff on the screen using `print`. This is Python's way of accessing standard output, or stdout, and you probably have a handle on it by now. Call `print` with a string, stuff appears on the screen. So this section would be unnecessary...

...if there wasn't an additional, slightly more formulaic way of doing the same thing directly with the standard output stream `sys.stdout`. In particular, just like the `stdin` stream includes `readline` for inputting a line of text, `stdout` includes a method `write` for outputting text. Not a line of text - just text, a single string, without a trailing newline as is forced upon every `printed` output. In general, `print( strIng )` and `sys.stdout.write( strIng + "\n" )` are equivalent, although you have to be a bit careful about who's a string when. `print` will convert its argument to a string for you (although I've been discouraging you from allowing this); `write` will not, so use `str()` liberally as needed.

I should also (very briefly) introduce another way of producing output. There's a third standard I/O object called `sys.stderr`, the standard error or stderr stream. In most cases, standard error behaves exactly like standard output. You can call `sys.stderr.write` and stuff will appear on the screen (in Eclipse, it'll be in pretty red rather than plain old black or dark blue or whatever the normal color is). We'll see below why it's sometimes useful to have a way of displaying information that's independent of standard output. Conceptually, the way in which standard error differs is that it's used for exceptional information. This means that when your program scores particularly well on a standardized test such as the GRE, you should give it a gold star and display the information using `sys.stderr`. It also means that when your program does something naughty and for some reason can't complete its normal task, you should use standard error for, well, errors. Consider something like this:

```
import sys
dValue = float(sys.stdin.readline( ))
if not dValue:
    sys.stderr.write( "Can't divide by zero!\n" )
sys.stdout.write( str(1.0 / d) + "\n" )
```

On the other hand, standard error is also commonly used for displaying status information that's unrelated to its main task or output. The idea is that the program's main results and "normal" output should go to standard out;

status information, errors, progress messages, and things that are useful for debugging should go to standard error. If we wanted to see how much progress our factorial program was making, for example, we might write code that looked like this:

```
sys.stderr.write( "Beginning factorial\n" )
iFact = 1
for i in range( 2, iN + 1 ):
    if not ( i % 10 ):
        sys.stderr.write( "i is " + str(i) + "\n" )
    iFact *= i
print( iFact )
```

This pattern is extremely useful for verifying that your loops are actually running! If you don't put any output in a long, complicated `for` loop, you might run it, wait for a long time, and worry that it's not actually making any progress. On the other hand, printing something out on every single iteration might produce hundreds or thousands of lines of output (it actually looks a lot like the screens on The Matrix). A test like the one above can be modified to produce one line of output for each 100 iterations, or each million, depending on the expected scale of your loop. Oh, and by the way, it's completely safe to mix `sys.stdout.write`, `sys.stderr.write`, and `print`, too.

1.3 Console I/O

> *"Bypasses are devices that allow some people to dash from point A to point B very fast while other people dash from point B to point A very fast. People living at point C, being a point directly in between, are often given to wonder what's so great about point A that so many people from point B are so keen to get there, and what's so great about point B that so many people from point A are so keen to get there." - "The Hitchhiker's Guide To The Galaxy"*

Ok, great. We have a console, we type stuff in, stuff comes out. Why is this interesting? Because, to reiterate one of the key lessons of these notes, we're lazy. We want to type as little as possible; we even want to read as little as possible. We want the computer to do all of the hard work for us. Suppose that we have a Python program that produces the first n digits of π for us. We run it on the command prompt:

```
C:\Documents and Settings\eblis>python pidigits.py 1000
3
1
4
...
```

We also happen to have a program that reads n numbers from standard input and then displays their sum:

```
C:\Documents and Settings\eblis>python sum.py 3
1
3
5
9
```

What if they want to merge their forces in the battle against evil, boredom, and mathematical knowledge and calculate the sum of the first 1000 digits of π? Well, you could print out all 1000 digits and then type them back in again. But that would be lame. Instead, the command prompt provides something called a redirect that will automatically:

- Read a file into a program's standard input, just as if you'd typed it on the console.
- Write a program's standard output into a file instead of the console.
- Read program A's standard output directly into program B's standard input.

This last one looks like what we want. When a redirect is used to connect one program's output to another program's input, it's called a pipe, and that's exactly the character to use:

```
python pidigits.py 1000 | python sum.py 1000
4473
```

You can pipe together as many programs as you want, sending each one's standard output along to the next one's standard input. For example, if we had a third program that just computed a square root, we could add:

```
python pidigits.py 1000 | python sum.py 1000 | python sqrt.py
66.880
```

Note that during this process, standard error still displays to the console! If we have progress messages every 500 iterations in both `pidigits.py` and `sum.py`, for example, we might get something like:

```
python pidigits.py 1000 | python sum.py 1000 | python sqrt.py
Calculating pi digit 0
Calculating pi digit 500
Summing number 0
Summing number 500
66.880
```

Redirecting standard input from a file works essentially the same was using the less than (<) character. Suppose we have a file containing 1000 numbers; to compute their sum without any extra typing, we could run:

```
python sum.py 1000 < file_o_numbers.txt
Summing number 0
Summing number 500
8675309
```

Again, standard error continues showing up on our console. If you flip your redirect around to get a greater than (>) character, you can dump standard output into a file instead:

```
python pidigits.py 3 > FileContaining314.txt
Calculating pi digit 0
```

Standard error is *still* there! It's tough to get rid of; that's why we like it for debugging. If your program's doing something goofy, putting in some extra `sys.stderr.write` calls to display pertinent information is often a good way to see what's going on.

You can combine input redirects, output redirects, and pipes in arbitrary ways to do funky stuff on the command line. It usually doesn't pay to get too fancy, but you could do something like:

```
python sum.py 100 < some_more_numbers.txt | python sqrt.py > squirt.txt
Summing number 0
```

After this whole mess is done running, your `squirt.txt` file will contain the square root of the sum of the 100 numbers read from the `some_more_numbers.txt` file. Important points to remember:

- Redirecting to a file using > will OVERWRITE the contents of the target file!!! Never forget this one. You can lose data this way faster than you can imagine.
- Trying to use file redirects and pipes in ways that don't make sense will cause problems. For example, if you're piping standard input from another program, you can't also redirect a file to standard input. Similarly, you can't redirect standard output to a file and to a pipe at the same time.

Total Eclipse of the Heart

In this case, the news if you're using Eclipse isn't so good.

- If you want to redirect standard output from a program running in Eclipse, you're in luck! Before you run your program, in the Run dialog box (the one you get by choosing Run... from the Run menu), click on the Common tab on the far right. At the bottom, you can turn on a checkbox named File and choose a file to receive your program's output redirect.
- If you want to redirect standard input to a program running in Eclipse or use pipes, you're hosed. There's no good way to do this. But there's a bad way! Run your program the usual way. It'll sit there doing nothing, waiting for some input; you should see the Console at the bottom of the screen with its little red square box. Open up the file you want to redirect into standard input, select the whole thing, copy it, and paste it into the Console window. It's ugly, but it works.