

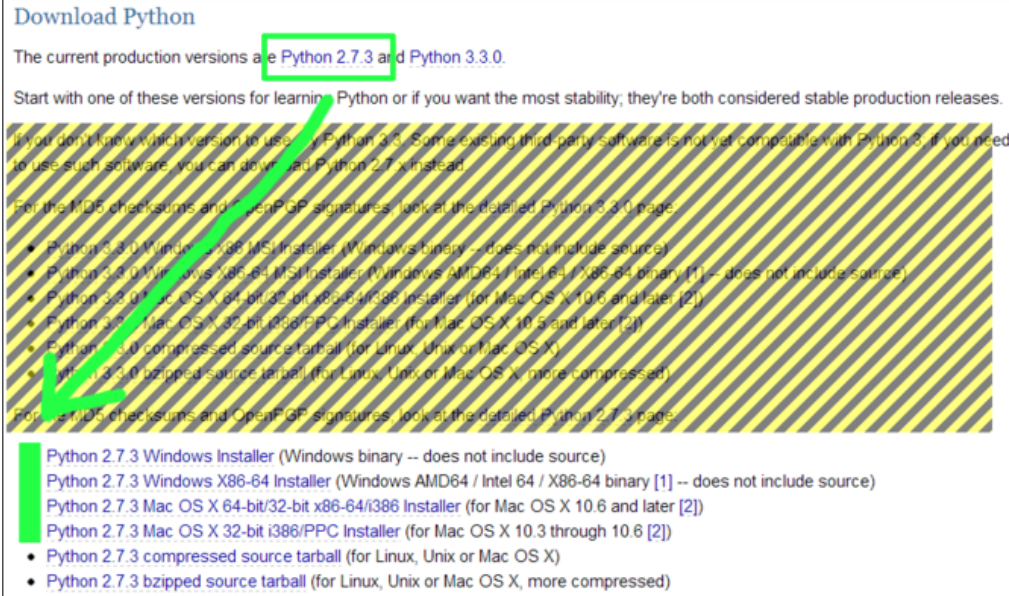
# Python language

## Install Python 2.7

You can skip this section if you are using a lab computer with Python already installed. Otherwise, here are your choices:

- Download Python 2.7.x from the [Python web site](#) (**not** version 3.y.z!), as shown in the rest of this section
- Use Python directly in your browser with [PythonAnywhere](#) – I have described how to get it working in [this short video](#).
- You may be able to use the [Python app](#) for the iPad – let me know how it works!

The rest of this section is about installing Python 2.7 on your computer.



The screenshot shows the 'Download Python' page. The text reads: 'The current production versions are Python 2.7.3 and Python 3.3.0.' The version '2.7.3' is highlighted with a green box. Below this, there is a large yellow and black striped warning area. A green arrow points from the '2.7.3' box down to the 'Python 2.7.3 Windows Installer' link in the list below the warning area. The list includes various installers for Windows, Mac OS X, and Linux/Unix.

**Download Python**

The current production versions are **Python 2.7.3** and Python 3.3.0.

Start with one of these versions for learning Python or if you want the most stability; they're both considered stable production releases.

If you don't know which version to use, try Python 3.3. Some existing third-party software is not yet compatible with Python 3, if you need to use such software, you can download Python 2.7.x instead.

For the MD5 checksums and OpenPGP signatures, look at the detailed Python 3.3.0 page.

- Python 3.3.0 Windows x86 MSI Installer (Windows binary -- does not include source)
- Python 3.3.0 Windows x86-64 MSI Installer (Windows AMD64 / Intel 64 / X86-64 binary [1] -- does not include source)
- Python 3.3.0 Mac OS X 64-bit/32-bit x86-64/i386 Installer (for Mac OS X 10.6 and later [2])
- Python 3.3.0 Mac OS X 32-bit i386/PPC Installer (for Mac OS X 10.5 and later [2])
- Python 3.3.0 compressed source tarball (for Linux, Unix or Mac OS X)
- Python 3.3.0 bzipped source tarball (for Linux, Unix or Mac OS X, more compressed)


For the MD5 checksums and OpenPGP signatures, look at the detailed Python 2.7.3 page.

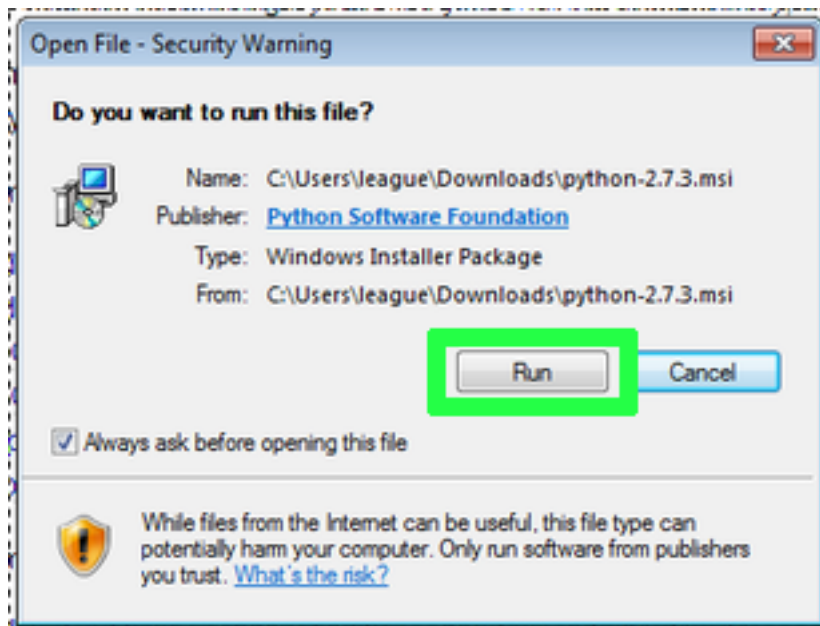
- **Python 2.7.3 Windows Installer** (Windows binary -- does not include source)
- **Python 2.7.3 Windows X86-64 Installer** (Windows AMD64 / Intel 64 / X86-64 binary [1] -- does not include source)
- **Python 2.7.3 Mac OS X 64-bit/32-bit x86-64/i386 Installer** (for Mac OS X 10.6 and later [2])
- **Python 2.7.3 Mac OS X 32-bit i386/PPC Installer** (for Mac OS X 10.3 through 10.6 [2])
- Python 2.7.3 compressed source tarball (for Linux, Unix or Mac OS X)
- Python 2.7.3 bzipped source tarball (for Linux, Unix or Mac OS X, more compressed)

To choose the right installer, you need to select between Mac and Windows, as well as between 32 and 64-bit. If you're not sure about the machine word size, from the Windows start menu you can choose Control Panel » System and Security » System, and look for the System type.

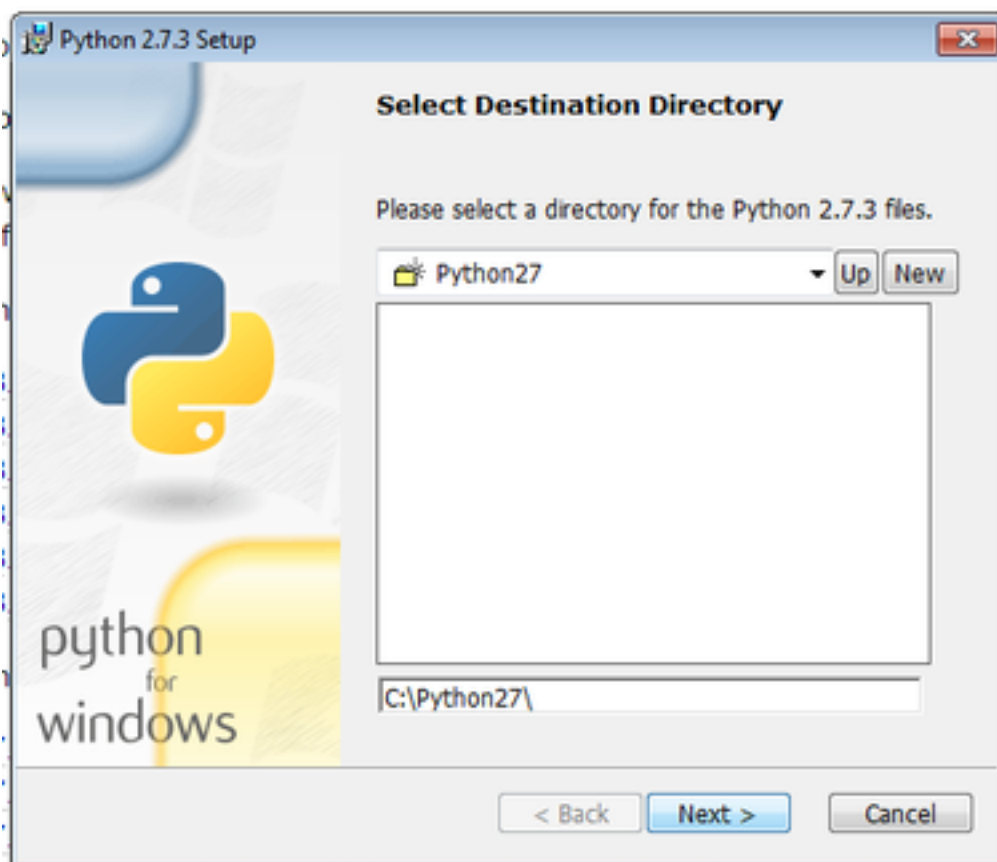
The install process is entirely straightforward, just work your way through the steps. I'll illustrate them below, but you probably won't need it.

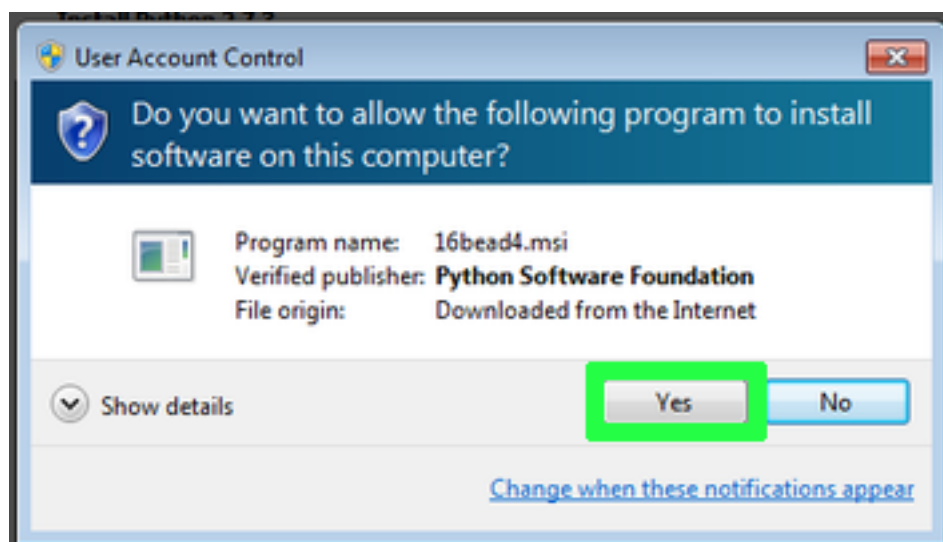
System

Rating:	 Windows Experience Index
Processor:	Intel(R) Core(TM) i7 CPU L 620 @ 2.00GHz 1.81 GHz
Installed memory (RAM):	1.76 GB
System type:	32-bit Operating System
Pen and Touch:	No Pen or Touch input is available for this Display





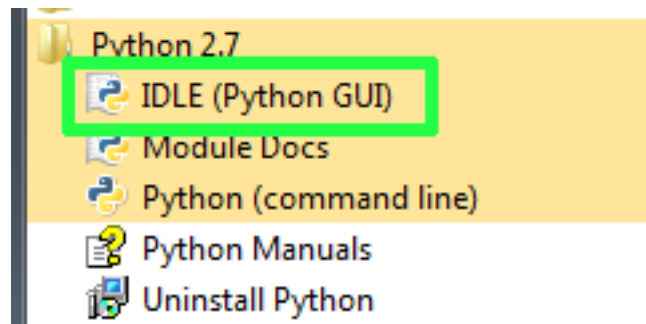




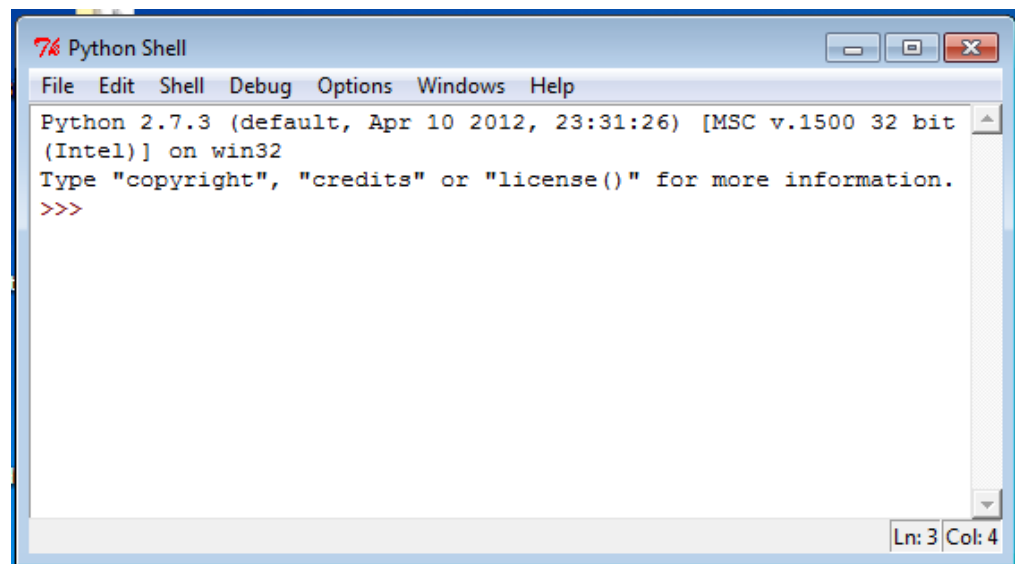


## Start your program

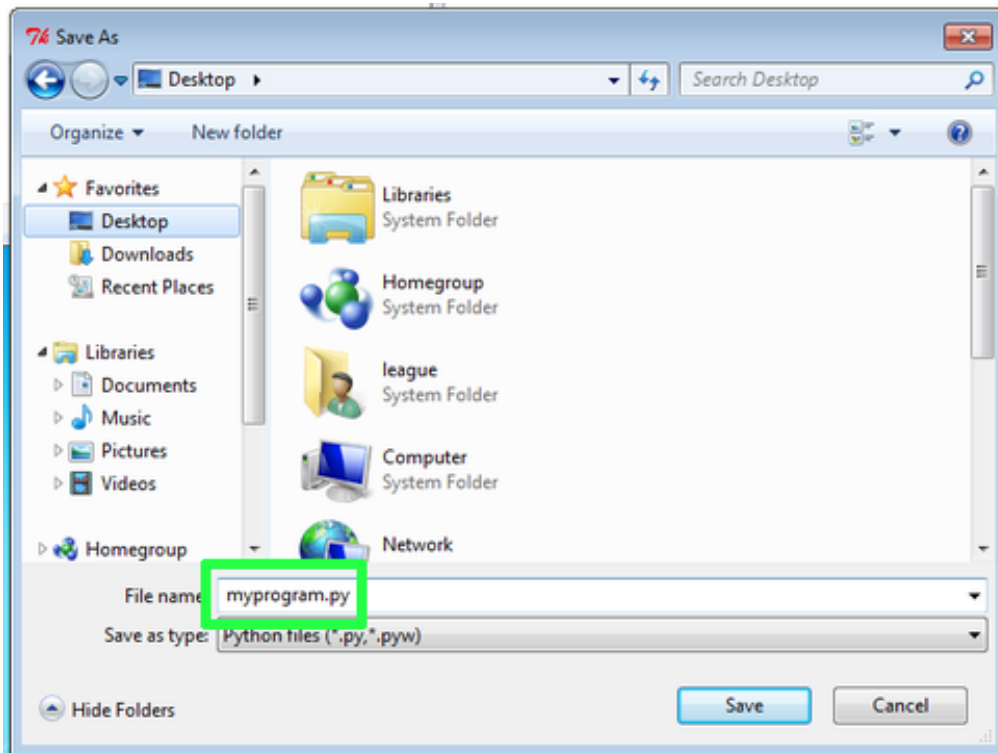
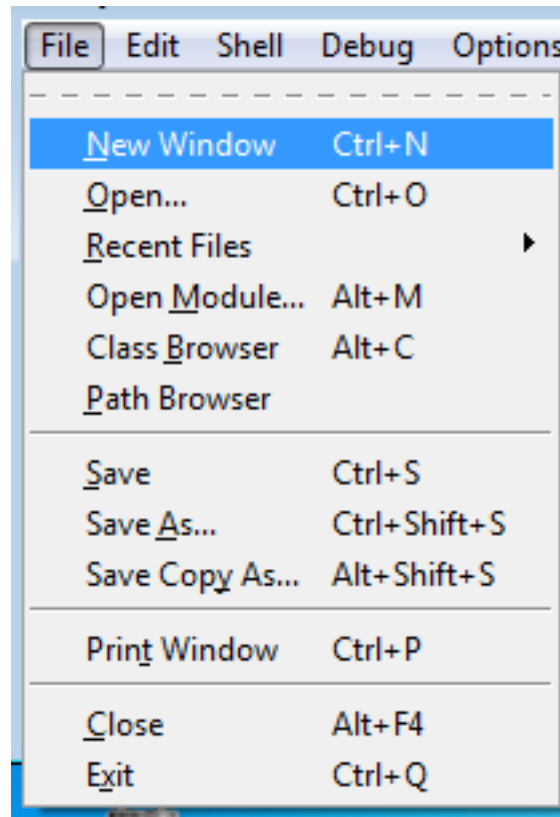
1. From the Windows Start menu, select Python 2.7 » IDLE (Python GUI). There will be a similarly named program in /Applications/Python 2.7 on the Mac.



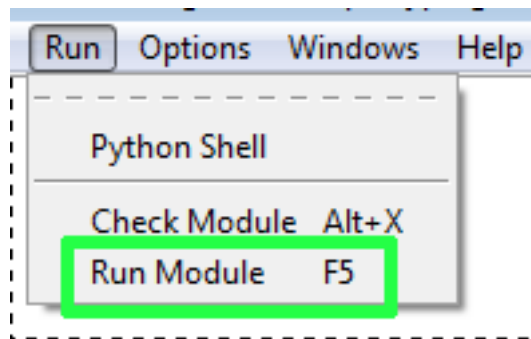
2. A window called the “Python Shell” will appear. It displays a prompt like “>>>”. This is the window in which you will interact with your program.



3. You need a second window, in which you will type and save the content of your program. Select File » New Window from the menu.
4. While in the new window, select File » Save and give it a filename that ends with .py (this will enable syntax coloring). You can also choose the folder to save it in, such as your Documents or Desktop folder.
5. Type your code into the .py file, and save it with File » Save, or Ctrl-S or ⌘-S.
6. Use Run » Run Module (F5) to run it, and interact with the program in the Shell window.







## Printing messages

Here is a program you can try that prints messages on the screen. You can copy and paste this block into your `.py` file, then save and run. (Note: the coloring used on this web page may not precisely match the colors you see in IDLE.)

```
print "Hello!"
print 3+4
print "3+4"
print "2*7 is", 2*7
```

When you run this program, the Shell window will contain:

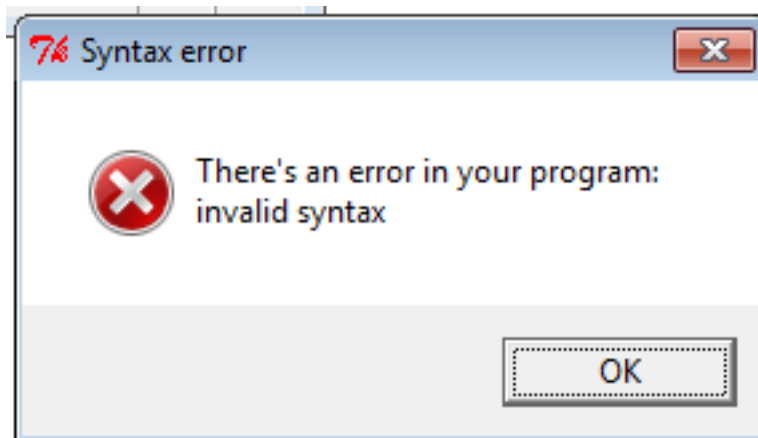
```
>>> ===== RESTART =====
>>>
Hello!
7
3+4
2*7 is 14
>>>
```

The double quotes indicate a *string* of characters (text). That portion is output exactly as written. Any portion *not* in quotes is interpreted by Python. Thus the difference between printing `3+4` (which produces `7`) and printing `"3+4"`.

## Fixing errors

The most common error you will see is probably “Invalid syntax”. It pops up like this: Dismiss that message, and you see it also highlights a portion of your program in red. Often the red part is just *after* the source of the actual error. In this case, we omitted the comma between the two different things in the print statement on that line.

Another kind of error shows up in the Shell window. It might look like this:



```
7% myprogram.py - C:/Users/league/Desktop/myprogram.py
File Edit Format Run Options Windows Help
print "Hello!"
print 3+4
print "3+4"
print "2*7 is" 2*7
```

The actual error is the last line. This one is called a `NameError`. But the Traceback section can help you too, by telling you which line number to examine (in this case, line 4).

## Doing calculations

We can store variables very simply in Python by using an equals sign. Variable names consist of a sequence of letters, numbers, and underscores (no spaces), but they cannot start with a number. Here are examples of valid variable names:

- `quiz3`
- `the_last_day`
- `Frank`
- `x`

These are **not** valid:

- `quiz-3` (contains a space)
- `2nd_quiz` (starts with a number)

Variable names are case-sensitive, so `x` and `X` are both valid, but they are not the *same* variable. Below is a program that uses variables to perform a computation.

```
quiz1 = 32
quiz2 = 40
quiz3 = 16
average = (quiz1 + quiz2 + quiz3) / 3.0
print "Average is", average
```

When you run it, the output should look something like this:

```
>>> ===== RESTART =====
>>>
Average is 29.3333333333
```

Python (and most languages) distinguish between numbers that are integers (whole numbers), and numbers that can contain decimal points. The latter are called floating-point numbers. When you divide two integers, Python performs *integer division*, which eliminates any remainder. You can overcome this by including a decimal point in your numeral, even if it's just `3.0` – that forces it to be represented as floating-point.

```
>>> 9/2
4
>>> 3/4
0
>>> 9/2.0
4.5
>>> 3.0/4
0.75
```

## Getting input

Python has a very handy built-in function for receiving *input* from the user of your program. It works like this:

```
name = raw_input("Enter your name: ")
```

On the left side of the equal sign is a variable name. On the right side is the `raw_input` function. In the parentheses, you specify a string that will be the *prompt* presented to the user. When you run the above program, it looks like this:

```
>>> ===== RESTART =====
>>>
Enter your name:
```

You are expected to type something at this point, and press enter. Whatever string you type will be placed into the variable name. Here's a more complete sample program:

```
name = raw_input("Enter your name: ")
print "Welcome to my program,", name

year = int(raw_input("What year were you born? "))
age = 2013 - year

print "You are", age, "years old."
```

Running the program looks like the following. The parts the user types are indicated by «angle quotes».

```
>>> ===== RESTART =====
>>>
Enter your name: «Chris»
Welcome to my program, Chris
What year were you born? «1988»
You are 25 years old.
```

## Boolean expressions

Python has values for Booleans – they are called True and False. Note that, like variable names, they are case-sensitive, so you must capitalize them as shown. There are also operators that *produce* Boolean values. The most obvious ones are for numeric comparisons. Consider this transcript in the Python shell:

```
>>> 3 < 5
True
>>> 3 > 5
False
>>> 2 < 2
False
>>> 2 <= 2
True
```

The last example in that block is <=, pronounced “less than or equal.” There is also >= for “greater than or equal.” You cannot have a space between the two operators: < = will be a syntax error.

Checking whether two things are exactly the same is a little tricky. The equals sign = is already used to mean *variable assignment*, as in:

```
my_quiz_score = 38
```

This statement above **does not** *ask* whether `my_quiz_score` is equal to the value 38. Instead, it **sets** the value of that variable to 38, and whatever value it had previously is lost.

In order to ask the question, whether a variable is equal to a certain value, you need to use a **double** equal sign: `==`, like this:

```
>>> my_quiz_score == 38
True
>>> 38 == my_quiz_score
True
>>> 21 == my_quiz_score
False
>>> 19 == 19
True
>>> 19 == 21
False
>>> "nice" == "evil"
False
>>> "nice" == "nice"
True
```

## Compound Booleans

Python also has operators from Boolean logic; they are called and, or, not. Here are a few examples involving them:

```
>>> 3 > 5 or 5 < 6    # becomes False or True, which is True
True
>>> 3 > 5 and 5 < 6   # becomes False and True, which is False
False
>>> not True
False
>>> not (3 > 5)
True
>>> my_quiz_score >= 0 and my_quiz_score <= 40
True
```

That last example determines whether the value of `my_quiz_score` is within a certain range: from zero to forty, inclusive.

## Conditional statement

Now that we've seen Boolean expressions, we're ready to explore conditional statements. You remember these from working with pseudo-code; they were statements like:

If  $X > 0$  then output  $X$  and stop.

The syntax in Python is a bit more regimented. There is a keyword `if` (must be lower case), and then the Boolean expression, followed by a colon (`:`) – it's a very common mistake to forget the colon!

On the next line, and indented a few spaces, you put any statements that should be executed **only if** the condition is true. Here is an example:

```
if my_quiz_score > 32:
    print "Congratulations, that's a good score."
    grade = "A"
print "Thanks for taking the course."
```

You can see that the *last* print statement in that block is not indented. That means it is no longer controlled by the `if`. If we reach this code when the value of `my_quiz_score` is 38, the output will be:

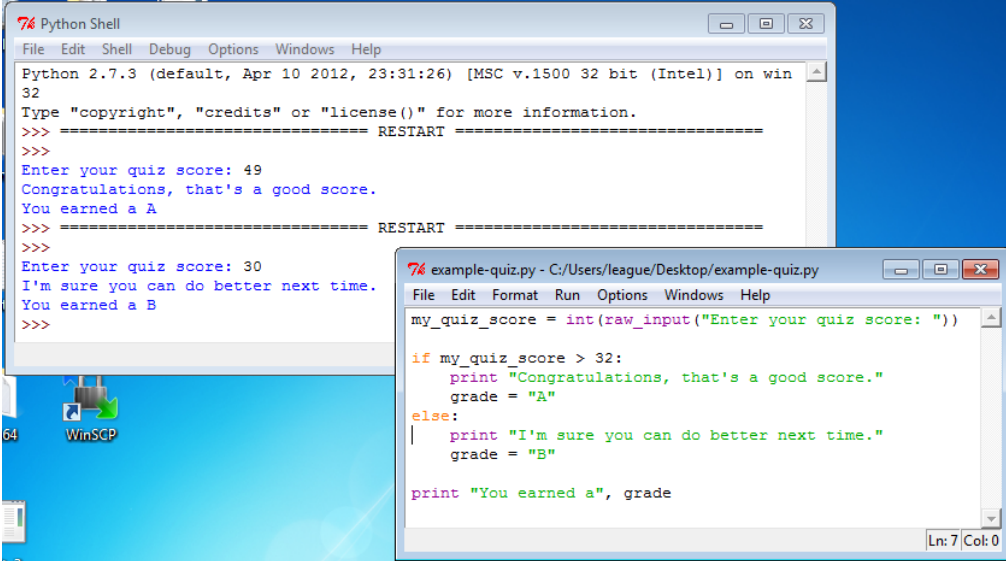
```
Congratulations, that's a good score.
Thanks for taking the course.
```

And the variable `grade` will contain the text string `"A"`. On the other hand, if `my_quiz_score` is 31, the output will be only the last line:

```
Thanks for taking the course.
```

It's also possible to provide an `else` block that executes when the `if` block doesn't. Here's a complete program you can paste into the Python program window and run with F5:

When you run it, try entering different values at the prompt in the shell window, and observe the results.



The image shows two windows from a Windows desktop. The top window is a Python Shell titled 'Python Shell'. It displays the following text:

```
Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Enter your quiz score: 49
Congratulations, that's a good score.
You earned a A
>>> ===== RESTART =====
>>>
Enter your quiz score: 30
I'm sure you can do better next time.
You earned a B
>>>
```

The bottom window is a text editor titled 'example-quiz.py - C:/Users/league/Desktop/example-quiz.py'. It contains the following Python code:

```
my_quiz_score = int(raw_input("Enter your quiz score: "))

if my_quiz_score > 32:
    print "Congratulations, that's a good score."
    grade = "A"
else:
    print "I'm sure you can do better next time."
    grade = "B"

print "You earned a", grade
```

## If/else chain

It's a fairly common pattern to “chain together” a series of if/else conditions, something like this:

```
if my_quiz_score > 32:
    grade = "A"
else:
    if my_quiz_score > 24:
        grade = "B"
    else:
        if my_quiz_score > 16:
            grade = "C"
        else:
            grade = "D"
```

You can see that the indentation increases each time an if is embedded within the else of another if. Try tracing this code with `my_quiz_score` set to different values, to see what ends up being stored in `grade`. You may even want to revise the previous program using this technique, so it can output more than just A or B.

This chain is common enough that Python provides a shortcut for it so that the continued indentation doesn't get out of hand. The shortcut relies on the keyword `elif`, and it looks like this:

```
if my_quiz_score > 32:
    grade = "A"
elif my_quiz_score > 24:
    grade = "B"
```

```
elif my_quiz_score > 16:  
    grade = "C"  
else:  
    grade = "D"
```

This program does the same thing as the previous one, but it's a little cleaner and shorter.

## Exercises

These are helpful exercises, to test putting it all together. First, create a program that does the following:

```
Enter quiz 1 score: «90»  
Enter quiz 2 score: «80»  
Your average is 85.0  
Your grade is B
```

It can just use 90/80/70/60 as the thresholds for A/B/C/D. Once you have that working, try this one:

```
Enter quiz 1 score: «60»  
Enter quiz 2 score: «95»  
Enter quiz 3 score: «92»  
Dropping the 60  
Your average is 93.5  
Your grade is A
```

It will ask for three scores, but then drop the lowest one before computing the average of the other two.

[\(My solution\)](#)

## Reopening your programs

Once you have saved your `.py` file, it may not work just to double-click your file again to reopen it. That will **run** the program in a console window, which is especially awkward because it might not pause to show the output before the window disappears.

Instead, you can right-click in the `.py` file and select "Edit with IDLE" (if on Windows). Or simply open IDLE first and then use the **File » Open** menu and navigate to the `.py` file from there.