Atom and Python with Hydrogen Package
This is my Atom setup for python -

use File > settings to set up your ATOM environment.
The Settings Tab:

from here you can set up your IDE
Select Packages to see what packages you have installed.

Select install to get the ones you want!
the packages I have...

busy-signal 1.4.3
A package that provides an easy to use API to show your package is performing a task
file-icons 2.1.16
Assign file extension icons and colours for improved visual grepping
highlight-selected 0.13.1
Highlights the current word selected when double clicking
intentions 1.1.5
Base package for showing intentions in Atom

linter 2.2.0
A Base Linter with Cow Powers
linter-flake8 2.3.0
Atom linter plugin for Python, using flake8
linter-ui-default 1.6.10
Default UI for the Linter package
I like having a minimap.
python-autopep8 0.1.3
Format python code using autopep8
markbaas

script 3.17.3
Run code in Atom!
rgbkrk

vim-mode-plus 1.27.0
vim-mode improved
t9md

Update to 1.32.0

annoying but really useful

hydrogen does this?

I have this cause vi is my default mode
to install what you want

all the stuff you can get.

Install Packages

- Packages are published to atom.io and are installed to C:\Users\etpe\atom\packages

Search packages

Featured Packages

- Hydrogen 2.4.1
  Run code interactively, inspect data, and plot. All the power of Jupyter kernels, inside your favorite text editor.

- atom-clock 0.1.16
  Display a customizable clock in the status bar.

- hey-pane 1.1.0
  Atom Plugin to enlarge the active pane, either with a shortcut or with a super handy follow mode. Inspired by Origami for Sublime Text.
Hydrogen

This package lets you run your code directly in Atom using any Jupyter kernels you have installed.

Checkout our Medium blog post to see what you can do with Hydrogen.

```
import matplotlib.pyplot as plt
import numpy as np
import sympy as sp
import pandas as pd

%matplotlib inline
%config InlineBackend.figure_format = 'svg'
sp.init_printing(use_latex='mathjax')

# Simple one line outputs
print('Hello World!')  # Hello World!
print('This is \$x_1b[00;38;33mHydrogen\$x_1b[0m:\$')  # This is Hydrogen.

# Simple plot inline figure
k = np.linspace(0, 2*np.pi, 50)
plt.plot(k, np.sin(k))
plt.show()
```
JavaScript
codeacademy

Build slideshow

Tuesday

WrapHTML

Fri

Project

RPS
Chapter 2. Flow Control

So you know the basics of individual instructions and that a program is just a series of instructions. But the real strength of programming isn’t just running (or executing) one instruction after another like a weekend errand list. Based on how the expressions evaluate, the program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. Flow control statements can decide which Python instructions to execute under which conditions.

These flow control statements directly correspond to the symbols in a flowchart, so I’ll provide flowchart versions of the code discussed in this chapter. Figure 2-1 shows a flowchart for what to do if it’s raining. Follow the path made by the arrows from Start to End.
Figure 2-1. A flowchart to tell you what to do if it is raining

In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles. The starting and ending steps are represented
Like any other value, Boolean values are used in expressions and can be stored in variables ❶. If you don’t use the proper case ❷ or you try to use True and False for variable names ❸, Python will give you an error message.
Comparison Operators

Comparison operators compare two values and evaluate down to a single Boolean value. Table 2-1 lists the comparison operators.

Table 2-1. Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>Equal to</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Not equal to</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

These operators evaluate to `True` or `False` depending on the values you give them. Let’s try some operators now, starting with `==` and `!=`.

```python
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

As you might expect, `==` (equal to) evaluates to `True` when the values on both sides are the same, and `!=` (not equal to) evaluates to `True` when the two values are different. The `==` and `!=` operators can actually work with values of any data type.

```python
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
```

*Case Sensitive*
True
>>> True != False
True
>>> 42 == 42.0
True
>>> 42 == '42'
False

Note that an integer or floating-point value will always be unequal to a string value. The expression `42 == '42'` evaluates to `False` because Python considers the integer `42` to be different from the string `'42'`.

The `<`, `>`, `<=`, and `>=` operators, on the other hand, work properly only with integer and floating-point values.

>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggCount = 42
>>> eggCount <= 42
True
>>> myAge = 29
>>> myAge >= 10
True

The Difference Between the `==` and `=` Operators
You might have noticed that the `==` operator (equal to) has two equal signs, while the `=` operator (assignment) has just one equal sign. It’s easy to confuse these two operators with each other. Just remember these points:

- The `==` operator (equal to) asks whether two values are the same as each other.
- The `=` operator (assignment) puts the value on the right into the variable on the left.

To help remember which is which, notice that the `==` operator (equal to) consists of two characters, just like the `!=` operator (not equal to) consists of two characters.

You’ll often use comparison operators to compare a variable’s value to some other value, like in the `eggCount <= 42` and `myAge >= 10` examples. (After all, instead of typing `'dog' != 'cat'` in your code, you could have just typed `True`.) You’ll see more examples of this later when you learn about flow control statements.
The three Boolean operators (and, or, and not) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value. Let’s explore these operators in detail, starting with the and operator.

**Binary Boolean Operators**

The and and or operators always take two Boolean values (or expressions), so they’re considered binary operators. The and operator evaluates an expression to True if both Boolean values are True; otherwise, it evaluates to False. Enter some expressions using and into the interactive shell to see it in action.

```python
>>> True and True
True
>>> True and False
False
```

A truth table shows every possible result of a Boolean operator. Table 2-2 is the truth table for the and operator.

*Table 2-2. The and Operator’s Truth Table*
**Truth Table**

All everything is true

<table>
<thead>
<tr>
<th>And</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

**OR**

<table>
<thead>
<tr>
<th>OR</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>V</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>V</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>V</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Unlike *and* and *or*, the *not* operator operates on only one Boolean value (or expression). The *not* operator simply evaluates to the opposite Boolean value.

```python
>>> not True
False
>>> not not not not True
True
```

Much like using double negatives in speech and writing, you can nest *not* operators 1, though there’s never not no reason to do this in real programs. **Table 2-4** shows the truth table for *not*.

### Table 2-4. The *not* Operator’s Truth Table

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to...</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>not True</em></td>
<td><em>False</em></td>
</tr>
<tr>
<td><em>not False</em></td>
<td><em>True</em></td>
</tr>
</tbody>
</table>
operators.

Recall that the *and*, *or*, and *not* operators are called Boolean operators because they always operate on the Boolean values *True* and *False*. While expressions like \(4 < 5\) aren’t Boolean values, they are expressions that evaluate down to Boolean values. Try entering some Boolean expressions that use comparison operators into the interactive shell.

\[
\begin{align*}
>>> & (4 < 5) \land (5 < 6) \\
& True \land True \\
>>> & (4 < 5) \land (9 < 6) \\
& False \land False \\
>>> & (1 == 2) \lor (2 == 2) \\
& False \lor True
\end{align*}
\]

The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value. You can think of the computer’s evaluation process for \((4 < 5) \land (5 < 6)\) as shown in *Figure 2-2*.

You can also use multiple Boolean operators in an expression, along with the comparison operators.

\[
\begin{align*}
>>> & 2 + 2 == 4 \land \neg 2 + 2 == 5 \land 2 * 2 == 2 + 2
\end{align*}
\]
```python
if name == 'Mary':
    # first block
    print('Hello Mary')
    if password == 'swordfish':
        # second block
        print('Access granted.')
    else:
        # third block
        print('Wrong password.')
else:
    # fourth block
```

The first block of code ❶ starts at the line `print('Hello Mary')` and contains all the lines after it. Inside this block is another block ❷, which has only a single line in it: `print('Access Granted.')`. The third block ❸ is also one line long: `print('Wrong password.')`. 
Logic for slideshow (JavaScript)
<html>
<head>
    <title>Banner</title>
</head>
<body>
    <div align="center">
        <img name="banner" src="images/failure-is-not-falling-down-proverb.gif">
    </div>
</body>
</html>
<script>
var imgArray = new Array();
var index = 0;
function cycle()
{
    document.banner.src = imgArray[index].src;
    index++;
    ⇒ index = index + 1;
    if (index > 5)
    {
        index = 0;
    }
    setTimeout("cycle()", 3000);
    return;
}