Intro

Recall that `True` and `False` are the two booleans in Python. It turns out they're pretty important. For instance, what if we want to do something, but only if a particular condition is `True`? What if we want to repeat something until that condition is `False`? These sorts of problems surface all the time — whether you're programming a calculator, a video game, a Terminator robot, or anything in between.

Comparisons

Now is a good time to learn about comparisons, which let us know which of two things is bigger or smaller. There are a few different comparisons that Python supports:

`>`	`x > y` if and only if `x` is bigger than `y`.
`<`	`x < y` if and only if `x` is smaller than `y`.
`>=`	`x >= y` if and only if `x` is greater than `y`, or `x` is equal to `y`.
`<=`	`x <= y` if and only if `x` is smaller than `y`, or `x` is equal to `y`.
`==`	`x == y` if and only if `x` is equal to `y`.
`!=`	`x != y` if and only if `x` is not equal to `y`.

Notice we use two equal signs instead of one, to test whether `x` equals `y`. That's because one equals sign is used for assigning variables, not testing equality. That means `x = y` and `x == y` are very different expressions.

```
>>> x = 3
>>> x
3
>>> x == 3
True
>>> x == 3.14
False
```

You can also compare several things at once. Every comparison must be `True` in order for the whole expression to evaluate to `True`. For instance, `x < y > z` requires `x < y` and `y > z`.

```
>>> 1 == 1 < 2 == 2
True
>>> 3.14 > 3 >= 2.9 + 0.1 == 3
True
>>> 3 > 2.5 > 1 < 1.0
False
```

```
>>> 4 != 5 != 5.0
False
>>> 3.14 <= 30 % 9 >= 2.718
False
>>> 3 > 2.5 > 1 <= 1.0
True
```

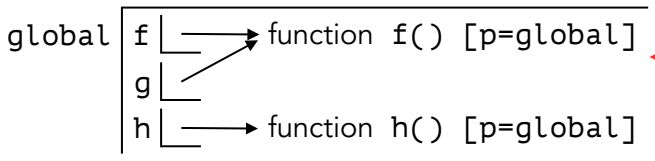
Comparison isn't just for numbers. Here's a table explaining all valid comparison you can make:

Integer `int`	Float `float`	Boolean `bool`	String `str`

You can compare amongst any integers, floats, and booleans. Replace `True` with a `1`, and `False` with a `0`. The rest works just like it would on a number line.
`>>> 7 == 7.0`
`True`
`>>> 3 > 1 == True > False > -1 <= 4`
`True`
			You can only compare strings to other strings.
			numbers < capitals < lower case
			`>>> "0"<"9"<"A"<"Z"<"a"<"z"`
			`True`

The `"=="` and `"!="` comparisons also work for variables that are assigned to pointers. Two pointers are equal if they point to the exact same thing, otherwise they are not equal. In the example below, `f == g` is `True` because `f` and `g` point to the same exact function. Meanwhile, `f` and `h` are actually different, because they point to two different functions. It doesn't matter that those two functions do the same thing.

```
>>> def f():
    return 5
>>> g = f
>>> def h():
    return 5
>>> f == g
True
>>> f == h
False
```



`f` and `g` are equal, since they're pointers at the same function. `f` and `h` are not equal, since they're pointing at different functions.

Conditions

This is big. We're about to unlock a new way to write code. You'll be able to do different things based on whether a condition is `True` or `False`. Check out these examples.

```
>>> def make_even(n):
...     if n % 2 == 0:
...         return n
...     return n + 1
>>> make_even(2)
2
>>> make_even(-5)
-4
```

It may seem strange that this function has two `return` statements. That's fine — remember, we automatically stop as soon as we hit either one. In this example we use the first one if `n` is even, otherwise we use the second one.

```
>>> def frog_latin(word):
...     if word < "frog":
...         return word + '-ay'
...     if word > "snake":
...         print('ribbit')
>>> frog_latin('armadillo')
'armadillo-ay'
>>> frog_latin('zebra')
ribbit
>>> frog_latin('zebra') == None
ribbit
True
```

You can also use the keyword `else`, which has to come right after an `if` statement. When you choose to have an `else` statement, your function will always execute either the `if` statement or the `else` statement but never both.

```
>>> def is_int(n):
...     if int(n) == n:
...         return True
...     else:
...         return False
>>> is_int(4)
True
>>> is_int(-10.3)
False
```

```
>>> def who_won(score):
...     if score > 0:
...         print('Cal')
...     else:
...         return 'Stanfurd'
>>> who_won(93)
Cal
>>> who_won(-10000)
'Stanfurd'
```

This is cool so far, but it gets even cooler. When you have a lot of different conditions, and you only want to use one of them at a time, you can use `elif` statements, short for "`else if`". You can have a bunch of `elif` statements in a row, if you want.

```
>>> def sign(n):
...     if n < 0:
...         return '-'
...     elif n > 0:
...         return '+'
...     else:
...         return 0
>>> sign(-1.1)
'-'
>>> sign(0) + 5
5
```

```
>>> def rock_type(rock):
...     if rock == "blotchy":
...         print('sedimentary')
...     elif rock == 'stripes!'
...         print('metamorphic')
...     elif rock == 'awesome':
...         return 'igneous'
...     return 'rocks rock!'
>>> rock_type('stripes!')
metamorphic
'rocks rock!'
>>> rock_type('octopus')
'rocks rock!'
>>> rock_type('awesome')
'igneous'
```

Be careful, `elif` can be tricky. It is *not* the same thing as using a bunch of `if` statements! That's because it's impossible to execute more than one statement in an `if` / `elif` chain, but it is certainly possible to execute more than one statement in a bunch of `if` statements.

```
>>> def size(n):
...     if n > 1000:
...         print('big')
...     if n > 0:
...         print('meh')
>>> size(2000)
big
meh
```

`2000` satisfies both conditions, so we execute both `if` statements.

```
>>> def size(n):
...     if n > 1000:
...         print('big')
...     elif n > 0:
...         print('meh')
>>> size(2000)
big
```

Since this is an `if` / `elif` chain, we just go with the first one that works and skip the rest.

You can also write `if` / `else` statements in one line. These two statements are identical:

```
>>> if 1 > 2:
...     'hi'
... else:
...     'bye'
'bye'
```

```
>>> 'hi' if 1 > 2 else 'bye'
'bye'
```

Don't do this too much. It gets really hard to read.

One last thing to know about conditions is that they are very lazy. In other words, they only evaluate something if they absolutely have to. This goes for one-line `if` / `else` statements too.

```
>>> def lucky(n):
...     if n % 10 != 7:
...         return 1 / 0
...     elif n == 13:
...         return '' + 5
...     else:
...         print('omg')
>>> lucky(37)
'omg'
```

When we run `lucky(37)`, these conditions evaluate to `False`. That means there's no need to evaluate the line that says `"return 1 / 0"` or the line that says `"return '' + 5"`, so we don't actually get an error. Instead, we just keep going until we find a condition that works.

```
>>> def uh_oh():
...     if 1 / 0 == 5:
...         return 5
>>> uh_oh()
Error
```

This is a problem because we always need to evaluate the condition itself, to see if it's `True` or `False`. Here, the condition causes an error.

Boolean Contexts

So far we've only seen conditions handle expressions that evaluate to `True` or `False`.

```
>>> if [          ]:
```

This should evaluate to `True` or `False`.

When the condition contains something that doesn't evaluate to `True` or `False`, we have a problem. Python solves that problem by automatically converting it to a boolean with the `bool(__)` function.

```
>>> if 'hi':
...     print('aliens')
aliens

>>> if '':
...     print('robots')
```

```
>>> if 3.14:
...     print('pirates')
pirates

>>> if print('dinosaurs'):
...     print('vs ducks')
dinosaurs
```

```
>>> bool('hi')
...     True
>>> bool('')
...     False
```

```
>>> bool(3.14)
...     True
>>> bool(None)
...     False
```

This doesn't mean any of those values *equal* `True` or `False`.

```
>>> 'hi' == True
False
```

```
>>> None == False
False
```

`1` and `0` are exceptions. That's because computers only understand binary, so `True` is really just another name for `1` and `False` is really just another name for `0`.

```
>>> 1 == True
True
```

```
>>> 0 == False
True
```

Boolean Operators

There are three boolean operators.

`not` (1)	`>>> not True` `False`	`>>> not False` `True`	`>>> not not True` `True`
`and` (2)	`>>> True and False` `False`	`>>> True and True` `True`	`>>> False and False` `False`
`or` (3)	`>>> True or False` `True`	`>>> True or True` `True`	`>>> False or False` `False`

These operators have to be applied in the exact order they're listed in the table: `not`, `and`, `or`. Take a look at the statements below. The top two match each other because it doesn't matter that we force `not` to be applied first. It is first by default, anyways! The statement on bottom does not match because we force `and` to be applied first, instead.

```
>>> not True and False
False
```

```
>>> (not True) and False
False

>>> not (True and False)
True
```

Now take a look at the next set of statements. The top two match each other because it doesn't matter that we force `and` to be applied before `or`. It comes before `or` anyways! The statement on bottom does not match because we force `or` to be applied before `and`, instead.

```
>>> False and False or True
True
```

```
>>> (False and False) or True
True

>>> False and (False or True)
False
```

You can use values other than booleans too. Like before, Python treats things like `True` if `bool(__)` returns `True`, and it treats things like `False` if `bool(__)` returns `False`.

```
>>> 'hi' and False
False
```

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

But there's a catch! Check out the next examples.

```
>>> 'hi' or False
'hi'
```

```
>>> 5 and 0
0
```

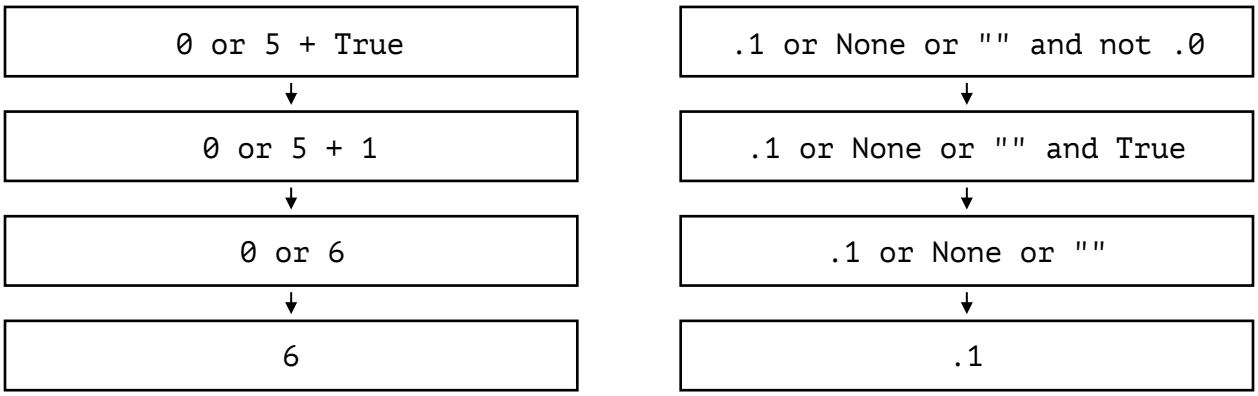
```
>>> True or False
True
```

```
>>> 5 and False
False
```

`'hi'` isn't literally the same as `True`, and `0` isn't literally the same as `False`. That means we're not allowed to replace `'hi'` with `True`, and we're not allowed to replace `False` with `0`. Instead of evaluating to `True`, the expression on the left now evaluates to `'hi'`. Similarly, instead of evaluating to `False`, the expression on the right now evaluates to `0`. Here are some more examples.

```
>>> None and True
None
```

```
>>> 'happy' or None
'happy'
```



Make sure you understand this before moving on. It may take a few minutes to fully understand, because it's a sort of tricky idea. Reread it or take a 5 minute break if that helps.

Take a look at these two examples.

```
>>> None or 0
0
```

```
>>> None and 0
None
```

The one on the left returns `0`, but the one on the right returns `None`. In order to understand why, we have to understand how `and` and `or` work.

`and` checks that every value is associated with `True`. If it comes across a value associated with `False`, it can stop right there. It returns the value of the very last thing it looks at.

In this example, `and` reaches the value `None`, which is associated with `False`. It stops there and returns `None`. It doesn't matter that later values would cause an error.

```
>>> -3 and True and "cake" and None and 8 and 'wheat' and 1/0 and LOL
None
```

In this example, every value is associated with `True`. `and` returns `"era"` since that's the last one it looks at.

```
>>> 'gecko' and 42 and "nest 'eggs'" and 18%5 and int(90.1) and "era"
"era"
```

`or` checks at least one value is associated with `True`. If it comes across a value associated with `True`, it can stop right there. It returns the value of the very last thing it looks at.

In this example, `or` reaches the value `7`, which is associated with `True`. It stops there and returns `7`. It doesn't matter that later values would cause an error.

```
>>> 0.0 or None or 18%9 or print(':)') or "" or 7 or 'one'*0 or 1/0
7
```

In this example, every value is associated with `False`. `or` returns `0.0` since that's the last one it looks at.

```
>>> int(0.9) or "one"*0 or '' or 2017%5-2 or bool(int(3.14-3)) or 0.0
0.0
```

Let's see some examples.