Recall `True` and `False` are the two booleans in Python. It turns out they're pretty handy. 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 come up 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 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 bigger 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 used two equals signs instead of one, to test whether `x` equals `y`. That's because one equals sign is used for assigning variables, not testing equality, so `x == y` is very different from `x = y`.

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

You can also make several comparisons at once, as in the examples below. Every individual 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` to both be `True`.

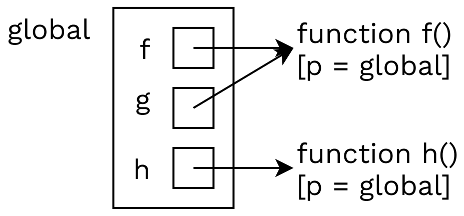
```
>>> 1 == 1 < 2 == 2
True
>>> 3.14 <= 30 % 9 >= 2.718
False
```

capital letters, which are less than strings of lowercase letters.

```
>>> '0' < '9' < 'A' < 'Z' < 'a' < 'z'
True
```

The `==` and `!=` operators work for pointers at functions, too. 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 exact same function. Meanwhile `f == h` is `False` because `f` and `h` 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
```



Conditions

This is where things get exciting. We're about to see how to write code that makes decisions.

The `if` and `else` statements

Using the `if` keyword, you can write code that does different things based on whether a condition is `True` or not. In particular, if the condition is `True`, then everything inside the `if` statement will happen. Otherwise it won't. Then, once

```
>>> def example(n):
...     if n == 0:
...         print('zero') # This happens when n is 0.
...     return n          # This always happens.
...
>>> example(0)
zero
0
>>> example(1)
1
```

It may seem strange, but this could lead to some situations when you have multiple `return` statements in one function. That's fine — remember, we automatically stop as soon as we hit either one. In the next example we use the first one if `n` is even, otherwise we use the second one.

```
>>> def is_even(n):
...    if n % 2 == 0:
...        return True
...    return False
...
>>> is_even(2)
True
>>> is_even(-5)
False
```

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 of the `else` statement but never both.

```
>>> def example(n):
...     if n == 0:
...         print('zero')    # This happens when n is 0.
...     else:
...         print('nonzero') # This happens when n isn't 0.
...     return n             # This always happens.
...
>>> example(0)
>>> example(1)
nonzero
1
```

Here's an example of how to write an `if` / `else` statement on one line. It's called a ternary if.

```
>>> forecast = 50
>>> 'snow' if forecast < 30 else 'rain'
'rain'
```

Try not to do this too often, though. It can get really hard to read.

The `elif` statement

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, 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
```

Below you'll find an equivalent way of expressing the function above, using only `if` and `else` statements. As you can see, it's a lot harder to read.

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

```
...         else:
...             return 0
...
>>> sign(-1.1)
'-'
>>> sign(0) + 5
5
```

Be careful; `elif` is sometimes tricky. It is *not* the same as `if`! Why? In a bunch of `if` statements, you might end up executing more than one. In an `elif` chain, on the other hand, you only execute the first statement whose condition is `True`. For example, compare the two definitions of the `size` function below.

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

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

Lazy evaluation

One last thing to know about conditions is that they are lazy. In other words, they only evaluate something if they absolutely have to. This goes for one-line `if` / `else` statements too. When we run `lucky(4)` in the code below, both conditions evaluate to `False`. That means there's no need to evaluate the part that says `return 1 / 0`

```
>>> def lucky(n):
...     if n == 7:
...         return 1 / 0
...     elif n == 13:
...         return 'zebra' + 10
...     else:
...         return True
...
>>> lucky(4)
True
```

Boolean operators

There are three boolean operators: `not`, `and`, and `or`. Let's see how they work.

1. `not` negates a boolean expression.

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

2. `and` requires that both boolean expressions are `True`.

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

3. `or` requires that either of two boolean expressions is `True`.

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

The order of evaluation

This is a problem, because some people might read it as $(1 + 2) \times 3$ and get 9, whereas others might read it as $1 + (2 \times 3)$ and get 7. In order to avoid confusion, there's a well-established convention that you should do multiplication first, and addition second. So the correct way to read $1 + 2 \times 3$ is like $1 + (2 \times 3)$.

We face a similar problem with the boolean operators. For instance, does `not True and False` mean `(not True) and False` or does it mean `not (True and False)`? Let's try it out in the Python interpreter and find out:

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

It looks like `not True and False` agrees with the output for `(not True) and False` but disagrees with the output for `not (True and False)`. That means `(not True) and False` is the right way to read it, which tells us `not` should come before `and`.

What about `or`? Let's try it out with `False and False or True`, which could mean `(False and False) or True` or `False and (False or True)`. Here's how we might test it out in the Python interpreter:

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

Since `False and False or True` matches with the output for `(False and False) or True`, we can see that `and` should come before `or`.

In summary, here's the proper order of boolean operations:

1. `not`
2. `and`
3. `or`

So far we've only seen boolean expressions that evaluate to either `True` or `False`. But what if we have a boolean expression that evaluates to something else? Like, what if we see `if 3.14: ...`? That might be an issue, since 3.14 is not a boolean. Python solves this problem by automatically converting it to a boolean with the `bool` function that we saw when we learned about the primitive types.

Here's an example. It will help to recall `bool(3.14)` and `bool('cowboys')` are both `True`, `bool('')` is `False`, and `bool(print('dinosaurs'))` is `False` because `print('dinosaurs')` evaluates to `None`.

```
>>> if 3.14:
...     print('vs pirates')
...
vs pirates
>>> if 'cowboys':
...     print('vs aliens)
...
vs aliens
>>> if '':
...     print('vs a vacuum')
...
>>> if print('dinosaurs'):
...     print('vs ducks')
...
dinosaurs
```

Things like 3.14 and `'cowboys'` are called truthy because you get `True` when you call `bool` on them. Similarly, things like `''` and `None` are called falsey because you get `False` when you call `bool` on them. They behave just like `True` and `False` whenever you're dealing with boolean expressions or `if` / `elif` / `else` conditions. But keep in mind, truthy values do *not* equal `True` and falsey values do *not* equal `False`.

```
>>> 3.14 == True
False
>>> None == False
False
```

The numbers 1 and 0 are exceptions. That's because everything inside your computer is represented by 1's and 0's, so fundamentally `True` is just another name for 1 and

```
>>> 1 == True
True
>>> 0 == False
True
```

Short-circuiting

Consider a boolean expression like `True or False`. It evaluates to `True`, but we didn't have to read the entire thing to figure that out. What if we stopped right after the `or`, so all we saw was `True or ____`? Whether we fill in the blank with `True` or `False`, this expression has to be `True`. (Try it out! `True or True` and `True or False` both evaluate to `True`.) So, if we already know the answer is going to be `True`, is there any reason to finish reading the whole expression? No, not really. In fact, Python stops reading as soon as it knows the answer. In this example, that's right before it would've read `False`. This is called short-circuiting.

Now is a good time to mention boolean expressions don't always evaluate to `True` or `False`. They evaluate to the last value Python reads before it knows the answer, which will always be a corresponding truthy or falsey value. For instance consider the expression `4 or False`. Since 4 is a truthy value, it's basically the same as our previous example `True or False`. Python doesn't have to finish reading it to know the expression is truthy. But in this case, it returns 4 instead of `True` because that's the last value Python had to read.

Let's take a closer look at short-circuiting with the `and` and `or` operators:

- `and` checks that every value is truthy, so if it comes across a falsey value it can stop right there. In the example below we reach the falsey value 0.0, where we stop and return 0.0. It doesn't matter that later values would cause an error.

  ```
  >>> -1 and 'cake' and 0.0 and 'is a' and 1 / 0 and lie
  0.0
  ```

  Now let's see an example where every value is truthy, so we just return `'glados'` since that's the last value Python looks at.

  ```
  >>> 'portal' and 42 != 54 and 7 < 8 < 9 and 'glados'
  'glados'
  ```

where we stop and return `'town'`. It doesn't matter that later values would cause an error.

```
>>> None or 18 % 9 or print('flavor') or 'town' or 'one' * 0 or 1
'town'
```

Now let's see an example where every value is falsey, so we just return 0 since that's the last value Python looks at.

```
>>> 'one' * 0 or int(0.9) or '' or 2018 % 5 - 3 or bool(int(3.14)
0
```

Evaluating boolean expressions

Now that we have a good understanding of the mechanics behind boolean expressions, we can make a procedure for how to evaluate them.

1. Add parentheses to the original expression, according to the order of boolean operations: `not`, `and`, then `or`.
2. Simplify the leftmost thing in your expression. If you can stop now, then it's your final answer. Otherwise delete it and repeat this step with your new, shorter expression.

Practice: miscellaneous boolean expressions

To make sure we have the hang of things, let's practice applying the procedure above. We'll start with the expression `'do' and 're' and 'mi' or None and not 0`.

1. First we add parentheses for `not`, to get `'do' and 're' and 'mi' or None and (not 0)`. Then we add parentheses for `and`, to get `('do' and 're' and 'mi') or (None and (not 0))`. It's not really necessary to add parentheses for `or` since it's the lowest-priority operator.
2. The leftmost thing in our expression is `'do' and 're' and 'mi'`. It's an `and` operator where every value is truthy, so Python reads the whole thing and evaluates it to `'mi'`. Now our expression looks like `'mi' or (None and (not 0))`. `'mi'` is truthy and we're evaluating an `or` operator, so we can stop here. Our final answer is `'mi'`.

1. First we add parentheses for `not`, to get `'a' and (not (not print('b'))) and 1 / 0 or 'c' + 123`. Then we add parentheses for `and`, to get `('a' and (not (not print('b'))) and 1 / 0) or 'c' + 123`. Again, it's not really necessary to add parentheses for `or` since it's the lowest-priority operator.

2. The leftmost thing in our expression is `'a' and (not (not print('b'))) and 1 / 0`. It's a bit complicated, so we'll apply our procedure to just this sub-expression in order to evaluate it.

    1. For step 1 in the procedure above, we just add the parentheses. Conveniently we've already done this, so we can move on.
    2. The leftmost thing in our sub-expression is `'a'`. It's truthy and we're on an `and` operator, so we have to keep going. According to the procedure above, we'll delete it and move on to `(not (not print('b'))) and 1 / 0`.
    3. Now the leftmost thing is `(not (not print('b')))`. `print('b')` returns `None` which is falsey, so `(not print('b'))` is `True` and `(not (not print('b')))` is `False`. Now our expression is `False and 1 / 0`. We can stop here and return `False`, because at this point our expression will be `False` no matter what comes next. Notice how we didn't have to evaluate `1 / 0`, which would've given us an error.

    Now that we have evaluated `'a' and (not (not print('b'))) and 1 / 0` to `False`, we can rewrite our bigger expression like `False or 'c' + 123`.

3. The leftmost thing is `False`, and since we're evaluating an `or` operator we have to keep going to see if there are any truthy values. According to the procedure above, we delete the `False` and keep going with just `'c' + 123`.

4. The leftmost thing is now `'c' + 123`, which causes an error. Whoops.