Lesson 2: More Operators and Some Conditionals

In this lesson, we cover quite a lot of ground, so buckle up.

Assigning & Changing Variables

We've talked a bit about *variables*, but the real power in variables is that that are variable, changeable. But so far we've mentioned variables in a static sense. So how do we change variables.

Let's start by directly changing it exactly how we want to:

In [1]:
```
num = 0 # this is an initiallization of the variable num (0 is a good starting point in ge
num = 2
print(type(num),num)
```

```
<class 'int'> 2
```

In [2]:
```
num = num + 1.5 # this changes num, increasing it by a fixed amount 1.5. this should also
print(type(num),num)
print(num)
num
```

```
<class 'float'> 3.5
3.5
```

Out[2]:
```
3.5
```

Hmmm, yes, nice. So, we can change the value of variables after assigning them. Now what? Well, now let's talk about *increment operators*.

Increment Operators

An increment operator updates the value of a variable whenever called (exectued).

In [3]:
```
x = 2
x += 4.1     # x = x + 4.1
x
```

Out[3]:
```
6.1
```

In [4]:
```
a # variable a hasn't been defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 a # variable a hasn't been defined

NameError: name 'a' is not defined
```

> **Note:** In order to directly outpute the value of any variable without calling the `print()` function, just type the variable at the end of your script/program and it will be outputted at the end of execution.

Basically, what's going on here is that we told the machine what `x` was, then told it to update `x`. And we can do this as many times as we like, which will be useful when we need to keep track of things.

In [5]:
```
x = 0 # initializing the variable x to be 0
x += 1 # x=1
x += 1 # x=2
x += 1 # x=3
x += 1 # x=4
x += 1 # x=5
x += 1 # x=6
x
```

Out[5]:
```
6
```

Type Conversion

Suppose you have a variable of one type, and you want to convert it to another. For example, say you have a string, `"42"`, and you want to convert it to an integer. This would happen if you were reading information from a text file, which by definition is full of strings, and you wanted to convert some string to a number. You would use something like the following code:

In [6]:
```
num_str = "42"
type(num_str)
```

Out[6]:
```
str
```

In [7]:
```
num = int(num_str)
print(num, type(num))
```

```
42 <class 'int'>
```

In [8]:
```
num_not = "3g"
numN = int(num_not)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[8], line 2
      1 num_not = "3g"
----> 2 numN = int(num_not)

ValueError: invalid literal for int() with base 10: '3g'
```

Simiarly, we can convert from `int` back to `str`:

In [9]:
```
str(num)
```

Out[9]:
```
'42'
```

Keep in mind that while `float` to `int` conversion is possible, a machine doesn't understand the concept of rounding as a human does. Way back in grade school, you were taught to round up and down. By default, a machine *always* rounds down:

In [10]:
```
int(3.7)
```

Out[10]:
```
3
```

And when combining strings and integers and conversions between, keep in mind the type you're working with when printing the output via `print()`.

In [11]:
```
print('4' + '2') # two strings being added
print(int('4') + int('2')) # two numbers being added
```

```
42
6
```

In [12]:
```
numA = '4' + '2'
numB = 4 + 2

print(type(numA),type(numB))
```

```
<class 'str'> <class 'int'>
```

Conditional Operators

Along with the good ol' arithmetic operators, machines also understand **conditional operators** which are used together to produce **conditional statements**.

"If you eat all your veggies, then you can have dessert." is a **conditional statement** of the *if-then* variety.

The operators we'll use to create conditional statements are **relational operators**, **identity operators**, and **logical operators**.

Relational Operators

Also called **comparison operators**, since **relational operators** are use to compare (or relate) two values/variables to one another. This could be testing if they are equal, if one is bigger than another, etc. The result of such an operation is either `True` or `False`, usually called a **Boolean** or *Bool* for short.

In [13]:
```
type(True)
```

Out[13]:
```
bool
```

In [14]:
```
type(False)
```

Out[14]:
```
bool
```

Every relational operator has an English equivalent you can use to read in place of the operator. See the table below.

Python	English
==	is equal to
!=	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

In [15]:
```
num = 5 # num is set to 5
num == 5 # "is num equal to 5?"
```

Out[15]:
```
True
```

In [16]:
```
num == 4
```

Out[16]:
```
False
```

In [17]:
```
5 < 4
```

Out[17]:
```
False
```

In [18]:
```
-17 > -209
```

Out[18]:
```
True
```

In [19]:
```
-17 >= -209
```

Out[19]:
```
True
```

In [20]:
```
'JamesMaxwell' == 'James Maxwell'
```

Out[20]: False

In [21]:
```
'James ' + 'Maxwell' != 'James Maxwell' # is blank NOT equal to Blank
```

Out[21]: False

Chaining relational operators

Python allos thew chaining of relational operators For example, we can test if a variable is within a certain range (useful for error analysis):.

In [22]:
```
1 < 2 < 4.6 < 15
```

Out[22]: True

In [23]:
```
1 < 2 < -4.6
```

Out[23]: False

One thing I'd warn you away from is mixing relational operators, even if a mixture is "legal" and true.

In [24]:
```
4 < 4.6 > 3
```

Out[24]: True

Identity operators

Identity operators check to see if two variables occupy the same space in memory; i.e., the're actually the same object (we'll go over objects later on in the bootcamp). This is stricltys different from the equality relational operato, `=` , which checks to see if two variables have the same value

| **Python** | **English** | |:---:|:---:| | `is` | is the same object | | `is not` | is not the same obtjec| | | | .

In [25]:
```
a = 4.2
b = 4.2

a == b, a is b
```

Out[25]: (True, False)

Notice how `==` and `is` result in different Boolean values, despite `a` and `b` having the same value. This is because the variables `a` and `b` are different, so are saved in different points of memory.

But what if we assign `b` to `a` directly?

In [26]:
```
a = 4.2
b = a
```

```
a == b, a is b
```

Out[26]: (True, True)

There we are! In this case, `==` and `is` give the same result! This is because we didn't just assign `b` to the same value as `a`, we assigned `b` to `a`, full stop. This means `b` takes the same place in memory as `a`.

What if we assign `b` to `a`, then change the value of `a`?

In [27]:
```
a = 4.2
b = a
a = 5.3

a == b, a is b, b is a
```

Out[27]: (False, False, False)

In this case, not only do `a` and `b` have different values, since we changed the value of `a` we also changed it's spot in memory. Thus, `b` no longer shares the placement of `a`.

There are, of course, exceptions to the rules, namely *integers*. In Python, we have to be careful when comparing integers between `-5` and `256`. This is because the Python language uses something called **integer caching**, which means integer objects are cached internally and reused via the same referenced objects. This saves memory, but can lead to some problems.

In [28]:
```
a = 4
b = 4

a == b, a is b
```

Out[28]: (True, True)

In [29]:
```
type(333)
```

Out[29]: int

In [30]:
```
a = 333
b = 333

a == b, a is b
```

Out[30]: (True, False)

Now, let's take a look at strings in this context.

In [31]:
```
a = 'hello world'
b = 'hello world'

a == b, a is b
```

Out[31]: (True, False)

As we saw in the case for `floats`, even though `a` and `b` have the same *value*, they don't have the same *location in memory*. Thus, they're different. Similarly to `floats`, assigning `b` to `a` changes that.

In [32]:
```
a = 'hello world'
b = a

a == b, a is b
```

Out[32]: (True, True)

In [33]:
```
a = 'python'
b = 'python'

a == b, a is b
```

Out[33]: (True, True)

As you can see from the above example, the rules don't always work the way you might expect them to. Again, this is because of a specific property of Python, called **string interning**, which means whether or not two strings occupy the same place in memory depends directly on what the strings are.

Generally speaking, we don't need to worry too much about *caching* and *interning* for **immutable** variables. **Immutable** means that once the variables are created, their values cannot be changed. And, if their values are changed, the variables get a new placement in memory to keep them distinct. So far, all the data types (`ints`, `floats`, and `strs`) are immutable.

Logical Operators

Logical operators can't be used in conjunction with the other operators, and, in Python, there are only three of them.

Python	Logic
`and`	AND
`or`	OR
`not`	NOT

The `and` operator means that if both operands are `True`, return `True`. The `or` operator gives

`True` if either of the operands are `True`. Finally, the `not` operator negates the logical result.

To see this in action, execute the following code blocks, but try to predict the result before you hit `shift+enter`.

In [ ]:
```
True and True
```

In [ ]:
```
True and False
```

In [ ]:
```
False and False
```

In [ ]:
```
True or False
```

In [ ]:
```
not False and True # True and True
```

In [ ]:
```
not(False and True) # not the same as "not False and not True"
```

In [ ]:
```
not False and not True # True and False
```

In [ ]:
```
not False or True # True or True
```

In [ ]:
```
not(False or True) # not(True)
```

In [ ]:
```
4 == 4 or 4.2 == 5.3 # True or False
```

In [ ]:
```
4 == 4 and 4.2 == 5.3 # True and False
```

Numerical Values of `True` & `False`

Even though these Boolean values output as strings, they are also associated with numerical values. And we can see this using some relational operators.

In [34]:
```
True == 1
```

Out[34]:
```
True
```

In [35]:
```
False == 0
```

Out[35]:
```
True
```

This means we can do arithmetic operations on Boolean values! However, before you get too excited, there's an implicit type conversion, as you can see in the below example.

In [36]:
```
type(True)
```

Out[36]:
```
bool
```

In [37]:
```
True + True, type(True + True)
```

Out[37]:
```
(2, int)
```

Conditionals

Now that we've covered all the operators, we can talk more freely about conditional statements. **Conditionals** are used to tell a machine to do a set of instructions depending on whether or not a Boolean is `True`. At the very beginning of this lesson, we mentioned `if-then` conditionals, so let's return to that example.

In [38]:
```
if something is true:
    do task(1)
else:
    do task(2)
```

```
  Cell In[38], line 2
    do task(1)
       ^
SyntaxError: invalid syntax
```

> **Note:** This is not *exactly* proper Python code, so when executed it will return an error (go ahead and check for yourself and see the note below). But it *is* close enough in general for study.
>
> For example: note the indentation of every other line. That is to signify that the indented line belongs to one results (`something is True`), while the next indented line belongs to the other possibility (`something is False`).
>
> > **SyntaxError:** A `SyntaxError` is reported by the Python interpreter when some portion of the code is incorrect. This can include misspelled keywords, missing or too many brackets or parentheses, incorrect operators, missing or too many quotation marks, or other conditions.

`if` Statement

Let's have a few more formal definitions. The Python `if` statement is used to determine the execution of code based on the evaluation of a Boolean expression.

If the `if` statement expression evaluates to `True` , then the indented code following the statement is execute . If the expression evaluates to `False` then the indented code following the `if` statement is skipped and the program executes the next line of code which is indented at the same level as the `if` statement.

`else` Statement

The Python `else` statement provides alternate code to execute if the expression in a previous `if` statement evaluates to `False`. The indented code for the `if` statement is executed if the expression evaluates to `True`. The indented code immediately following the `else` is executed only if the expression evaluates to `False`. To mark the end of the `else` block, the code must be unindented to the same level as the starting `if` line.

In the case of the example above, there are only two possible routes: either `something is True` or `something is False`. But many times you want to be more flexible than that. That's where `if-elif` comes in handy, since you can include as many `elifs` as you need.

`elif` Statement

The Python `elif` statement allows for continued checks to be performed after an initial `if` statement. An `elif` statement differs from the `else` statement because another expression is provided to be checked, just as with the initial `if` statement.

If the expression is `True`, the indented code following the `elif` is executed. If the expression evaluates to `False`, the code can continue to an optional `else` statement. Multiple `elif` statements can be used following an initial `if` to perform a series of checks. Once an `elif` expression evaluates to `True`, no further `elif` statements are executed.

In [ ]:
```
if something > 0:
    do task(1)
elif something < 0:
    do task(2)
elif something is 0:
    do task(3)
```

You can also use `if-else` statements within other `if-else` statements. This is called **nesting**.

In [ ]:
```
if something > 0:
    do task(1)
else:
    if something < 0:
        do task(2a)
    else:
        do task(2b)
```

Please proceed to Assignment 2.
