Booleans

Python has a type of variable called `bool`. It has two possible values: `True` and `False`.

In [1]:
```
x = True
print(x)
print(type(x))
```

```
True
<class 'bool'>
```

Rather than putting `True` or `False` directly in our code, we usually get boolean values from **boolean operators**. These are operators that answer yes/no questions. We'll go through some of these operators below.

Comparison Operations

Operation	Description	Operation	Description
`a == b`	`a` equal to `b`	`a != b`	`a` not equal to `b`
`a < b`	`a` less than `b`	`a > b`	`a` greater than `b`
`a <= b`	`a` less than or equal to `b`	`a >= b`	`a` greater than or equal to `b`

In [2]:
```
def can_run_for_president(age):
    """Can someone of the given age run for president in the US?"""
    # The US Constitution says you must be at least 35 years old
    return age >= 35

print("Can a 19-year-old run for president?", can_run_for_president(19))
print("Can a 45-year-old run for president?", can_run_for_president(45))
```

```
Can a 19-year-old run for president? False
Can a 45-year-old run for president? True
```

Comparisons frequently work like you'd hope

In [3]:
```
3.0 == 3
```

Out[3]:
```
True
```

But sometimes they can be tricky

In [4]:
```
'3' == 3
```

Out[4]:
```
False
```

Comparison operators can be combined with the arithmetic operators we've already seen to express a virtually limitless range of mathematical tests. For example, we can check if a number is odd by checking that the modulus with 2 returns 1:

In [5]:
```
def is_odd(n):
    return (n % 2) == 1

print("Is 100 odd?", is_odd(100))
print("Is -1 odd?", is_odd(-1))
```

```
Is 100 odd? False
Is -1 odd? True
```

Remember to use `==` instead of `=` when making comparisons. If you write `n == 2` you are asking about the value of n. When you write `n = 2` you are changing the value of n.

Combining Boolean Values

You can combine boolean values using the standard concepts of "and", "or", and "not". In fact, the words to do this are: `and`, `or`, and `not`.

With these, we can make our `can_run_for_president` function more accurate.

In [6]:
```
def can_run_for_president(age, is_natural_born_citizen):
    """Can someone of the given age and citizenship status run for president
    # The US Constitution says you must be a natural born citizen *and* at l
    return is_natural_born_citizen and (age >= 35)

print(can_run_for_president(19, True))
print(can_run_for_president(55, False))
print(can_run_for_president(55, True))
```

```
False
False
True
```

Quick, can you guess the value of this expression?

In [7]:
```
True or True and False
```

Out[7]:
```
True
```

(Click the "output" button to see the answer)

To answer this, you'd need to figure out the order of operations.

For example, `and` is evaluated before `or`. That's why the first expression above is `True`. If we evaluated it from left to right, we would have calculated `True or True`

first (which is `True`), and then taken the `and` of that result with `False`, giving a final value of `False`.

You could try to memorize the order of precedence, but a safer bet is to just use liberal parentheses. Not only does this help prevent bugs, it makes your intentions clearer to anyone who reads your code.

For example, consider the following expression:

```
prepared_for_weather = have_umbrella or rain_level < 5 and
have_hood or not rain_level > 0 and is_workday
```
I'm trying to say that I'm safe from today's weather....

- if I have an umbrella...
- or if the rain isn't too heavy and I have a hood...
- otherwise, I'm still fine unless it's raining *and* it's a workday

But not only is my Python code hard to read, it has a bug. We can address both problems by adding some parentheses:

```
prepared_for_weather = have_umbrella or (rain_level < 5 and
have_hood) or not (rain_level > 0 and is_workday)
```
You can add even more parentheses if you think it helps readability:

```
prepared_for_weather = have_umbrella or ((rain_level < 5) and
have_hood) or (not (rain_level > 0 and is_workday))
```
We can also split it over multiple lines to emphasize the 3-part structure described above:

```
prepared_for_weather = (
    have_umbrella
    or ((rain_level < 5) and have_hood)
    or (not (rain_level > 0 and is_workday))
)
```

Conditionals

Booleans are most useful when combined with *conditional statements*, using the keywords `if`, `elif`, and `else`.

Conditional statements, often referred to as *if-then* statements, let you control what pieces of code are run based on the value of some Boolean condition. Here's an example:

In [8]:
```
def inspect(x):
    if x == 0:
        print(x, "is zero")
    elif x > 0:
```

```
        print(x, "is positive")
    elif x < 0:
        print(x, "is negative")
    else:
        print(x, "is unlike anything I've ever seen...")

inspect(0)
inspect(-15)
```

```
0 is zero
-15 is negative
```

The `if` and `else` keywords are often used in other languages; its more unique keyword is `elif`, a contraction of "else if". In these conditional clauses, `elif` and `else` blocks are optional; additionally, you can include as many `elif` statements as you would like.

Note especially the use of colons (`:`) and whitespace to denote separate blocks of code. This is similar to what happens when we define a function - the function header ends with `:`, and the following line is indented with 4 spaces. All subsequent indented lines belong to the body of the function, until we encounter an unindented line, ending the function definition.

In [9]:
```
def f(x):
    if x > 0:
        print("Only printed when x is positive; x =", x)
        print("Also only printed when x is positive; x =", x)
    print("Always printed, regardless of x's value; x =", x)

f(1)
f(0)
```

```
Only printed when x is positive; x = 1
Also only printed when x is positive; x = 1
Always printed, regardless of x's value; x = 1
Always printed, regardless of x's value; x = 0
```

Boolean conversion

We've seen `int()`, which turns things into ints, and `float()`, which turns things into floats, so you might not be surprised to hear that Python has a `bool()` function which turns things into bools.

In [10]:
```
print(bool(1)) # all numbers are treated as true, except 0
print(bool(0))
print(bool("asf")) # all strings are treated as true, except the empty strin
print(bool(""))
# Generally empty sequences (strings, lists, and other types we've yet to se
# are "falsey" and the rest are "truthy"
```

```
True
False
True
False
```

We can use non-boolean objects in `if` conditions and other places where a boolean would be expected. Python will implicitly treat them as their corresponding boolean value:

In [11]:
```
if 0:
    print(0)
elif "spam":
    print("spam")
```

```
spam
```

Your Turn

You probably don't realize how much you have learned so far. Go try the **hands-on coding problems**, and you'll be pleasantly surprised about how much you can do.
