Contents

1 Booleans (`bool`)

In the last lab, you learnt about 3 data types: the integer, float, and string. A boolean is another data type, with only two possible values, to which we assign the values of 'true' and 'false'. In Python, the two boolean values are noted with capital letters - `True` and `False`.

```python
x = True
print(x)
y = False
print(y)
```

You can also use the `bool()` function to convert a value (`str`, `int`, `float`, or others) into a boolean, which you will experiment with in the exercises.

1.1 Boolean operators - `and`, `or`, `and not`

Just as you can add, subtract, multiply, and divide numbers, you can also perform operations on boolean values.

In Python, there are three boolean operators. They function the same way as in math or in everyday language. For clarity, how each operator evaluates is written in the truth table below, written in the proper syntax, where P and Q are booleans.

P	Q	P and Q	P or Q	not P
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

You can see these values for yourself in Python.

```python
print(True and False)    # False
print(False or False)    # False
print(not True)          # False
```

2 Relational operators

We typically see boolean values as the result of some comparison of values. These comparisons are performed through *relational operators*, listed below with their meaning. Hopefully they are familiar to you, from learning inequalities in mathematics.

Relational operators *are not statement of fact* about the expressions to the left and right of them, but more of a 'question of fact'. Is the LHS equal to to the RHS? Is the LHS greater than the RHS? And Python will respond with whether it is `True` or `False` for each question you ask.

Operator	Meaning
`==`	equal to
`!=`	not equal to
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to

Notably, the = operator is used for the assignment operation, and the == operator is used to test for equality. Again, you can experiment with the relational operators in Python.

```
x = 1
print("1" == x)      # False - they are not the same data type.
print(1 == x)        # True
print(2 != x)        # True
print("A" < "B")     # True - you can compare strings.
```

2.1 Chained relational operators

One bit of syntax relatively unique to Python is the ability to chain comparisons. Typically you would only chain the inequality operators, to see if something is between two values:

```
print(-1 < x < 1)   # equivalent to (-1 < x) and (x < 1)
```

3 Conditional statements

With the knowledge of booleans and where they frequently occur, we can move on to a new programming concept and Python syntax. Previously, all the code written was run line by line, without skipping any. Conditional statements allow us to *control the flow* of the program, by skipping or executing certain lines based on some *conditions*.

Run the following code blocks yourself, changing the values of `expression1` and `expression2` and compare what is printed carefully.

3.1 `if` statements

An `if` statement checks a condition and executes a block of code if the condition is true.

```
expression1 = True

if expression1:
  # code to run if expression1 is True
  print("I ran because expression1 was true.")
```

You may notice that there are some lines which are indented (not aligned to the left). This is where *indentation* becomes important in Python, as they define *what code to run*. The amount of indentation used throughout the labs is two spaces (others typically use four spaces, or a single tab character), but you are free to use any amount of indentation, as long as you are consistent within a block.

3.2 `if-else` statements

An `if-else` statement checks a condition and executes one of two blocks of code: the `if` block if the condition is true, or the `else` block if the condition is false.

Note that **no condition or expression** can be put after the `else` keyword.

```python
expression1 = True

if expression1:
  # code to run if expression1 is True
  print("I ran because expression1 was true.")
else:
  # code to run if expression1 is False
  print("I ran because expression1 was false.")
```

`if-else` statements can be nested to test for multiple conditions, though an `if-elif` statement is more clear to read.

```python
expression1 = True
expression2 = True

if expression1:
  # code to run if expression1 is True
  print("I ran because expression1 was true.")
else:
  print("I ran because expression1 was false.")
  if expression2:
    # code to run if expression1 is False and expression2 is True
    print("I ran because expression1 was false, and expression2 was true."
      )
  else:
    # code to run if both expression1 and expression2 are False
    print("I ran because expression1 was false, and expression2 was false.
      ")
```

3.3 `if-elif` statements

An `if-elif` (`elif` read as ‘else if’) statement checks a condition and executes the `if` block if the condition is true. If the condition is false, it checks another condition and executes the `elif` block if the second condition is true. Multiple `elif` statements can be used.

```python
expression1 = True
expression2 = True

if expression1:
  # code to run if expression1 is True
  print("I ran because expression1 was true.")
elif expression2:
  # code to run if expression1 is False and expression2 is True
  print("I ran because expression1 was false, and expression2 was true.")
```

3.4 `if-elif-else` statements

An `if-elif-else` statement checks multiple conditions and executes the first block of code that matches a true condition. If none of the conditions are true, it executes the `else` block. Multiple `elif` statements can be used, and the `else` statement *must* be the last statement of the conditional.

```python
expression1 = True
expression2 = True

if expression1:
  # code to run if expression1 is True
  print("I ran because expression1 was true.")
elif expression2:
  # code to run if expression1 is False and expression2 is True
  print("I ran because expression1 was false, and expression2 was true.")
else:
  # code to run if both expression1 and expression2 are False
  print("I ran because both expression1 and expression2 were false.")
```

3.5 Short-circuiting

In conditional expressions that use and or `or`, Python will stop evaluating the statement as soon as the final boolean outcome is determined.

For example, in an and operator, if either expression is false, the final expression is false. Thus, when the first expression is false, Python does not need to evaluate the second expression.

Likewise, in an `or` operator, if either expression is true, the final expression is true. So, if the first expression is true, Python does not need to evaluate the second expression.

```python
expression1 = True

if expression1 or 1 / 0:
  # True or (whatever)
  print("I can still run despite dividing by zero.")
```

3.6 `if-elif-else` example

```
x = -1                              # line 1
if x > 0:                           # line 2: x is -1, so x > 0 is False
  print("x is positive")            # line 3: not run, does not print
elif x < 0:                         # line 4: x is -1, so x < 0 is True
  print("x is negative")            # line 5: prints, short circuits
else:                               # line 6: not run
  print("x is 0")                   # line 7
print("the value of x is", x)       # line 8: prints
```

```
x is negative
the value of x is -1
```

Python reads the code line-by-line as usual, skipping blocks as directed by the conditional statements. In total, Python will evaluate 8 lines, which are:

1. line 1: define a variable `x`, set it to `-1`.
2. line 2: check if `x > 0`. Since `x` is `-1`, `x > 0` is `False`, so **skip** the block directly below it.
3. line 4: check if `x < 0`. Since `x` is `-1`, `x < 0` is `True`, so **enter** the block directly below it.
4. line 5: print the given expression. The conditional short-circuits and **exits** the `if-elif-else` statement completely.
5. line 8: print the given expression.

4 Additional arithemetic operations

In addition to `+`, `-`, `*`, and `/` (addition, subtraction, multiplication, division), there are several other operations which are useful.

4.1 Exponentiation (`**`)

Exponentiation in math is written as a^b. In Python it is written as `a**b`.

```
print("10 ** 3 is", 10 ** 3)    # 1000, since 10^3 = 1000
print("22 ** 5 is", 22 ** 5)    # 5153632, since 22^5 = 5153632
```

4.2 Modulo (`%`)

The next two operators are related. When you divide one number by another, you get a *quotient* and a *remainder*. The quotient is the result of the division, and the remainder is the leftover amount that does not fit into the quotient.

The modulo operator returns the *remainder* of a division. In Python, it is written as `a % b`, where `a` is the dividend, and `b` is the divisor.

```
print("10 % 3 is", 10 % 3)      # 1, since 10 / 3 = 3 **remainder 1**
print("22 % 5 is", 22 % 5)      # 2, since 22 / 5 = 4 **remainder 2**
```

4.3 Floor division (//)

The floor division operator is the counterpart to the modulo operator. It returns the *quotient* of a division. In Python, it is written as a //b, where a is the dividend, and b is the divisor.

```
print("10 // 3 is", 10 // 3)    # 3, since 10 / 3 = **3** remainder 1
print("22 // 5 is", 22 // 5)    # 4, since 22 / 5 = **4** remainder 2
```

4.4 Shorthand arithmetic assignment

This is not a new operator, per se, but a shorthand for when you want to reassign a variable after changing it with some arithmetic operator. You can do this for all the above operators, including exponentiation, modulo and floor division. For example, to increment a variable x by 1:

```
x = 0
x += 1     # equivalent to x = x + 1
```

5 Order of operations

We've taught you a lot of operators today. Like in math, there is a certain order in its execution, and in Python, it is not too different. Parentheses are always evaluated first. The arithmetic operators go next (in the order you would expect), followed by the relational operators, the boolean operators, and finally the assignment operator.

You can see the complete list in the Python documentation, though it covers several operators we have not covered.

Precedence	Operator	Meaning
highest priority	`(expressions...)`	parentheses
	`**`	exponentiation
	`*`, `/`, `//`, `%`	multiplication, division, floor division, remainder
	`+`, `-`	addition, subtraction
	`<`, `<=`, `>`, `>=`, `==`, `!=`	relational operators
	`not`	boolean NOT
	`and`	boolean AND
	`or`	boolean OR
lowest priority	`=`	assignment