The journey so far

- What is a computer?
- Data input
- Data output
- Data processing with operators
- Data types
- Basic Python syntax

Take control!

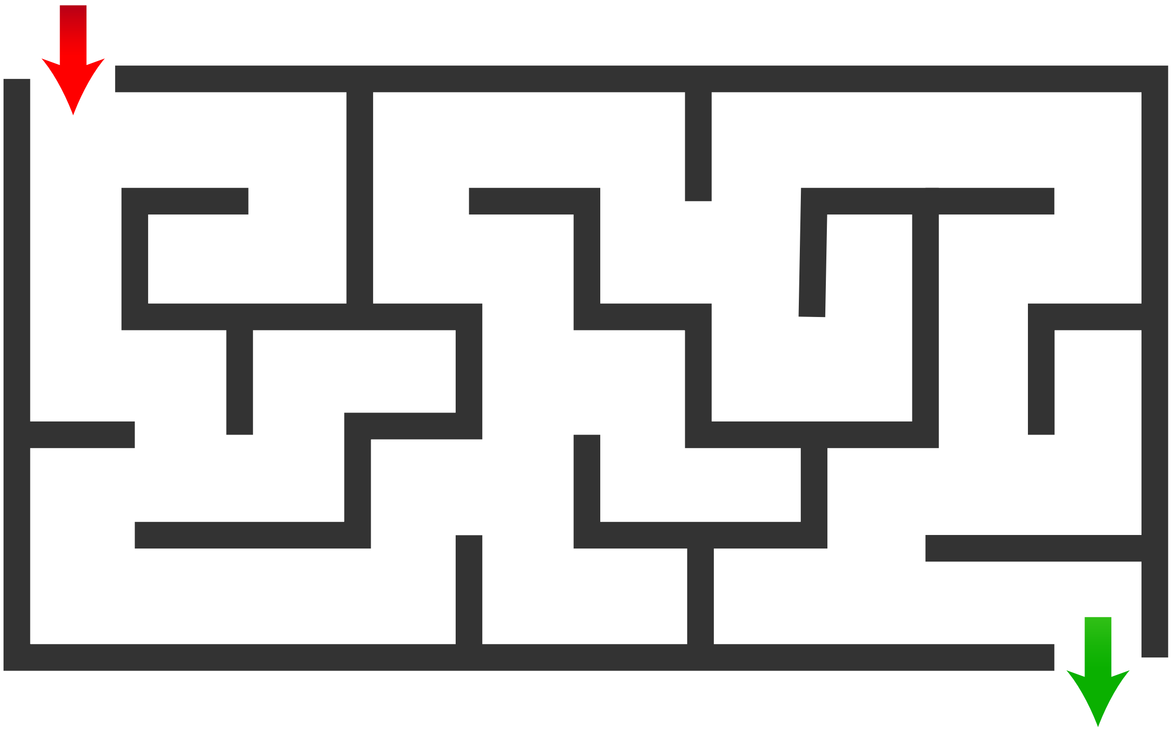
Computers are, in fact, incredibly stupid. They are completely incapable of taking any kind of decisions. Luckily, you are here to guide it.

In this notebook you'll learn a bit more about what a computer actually is and how we can use it to our advantage. Finally you'll extend your cookie dough program to become a little smarter...

Machines!

Machines can't really take any kinds of decision by themselves. The idea of computers as evil machines, is far from reality.

A better picture of a computer is like a maze. When we tell it to, it will start to execute a program (the maze) and then it starts taking a number of decisions. Some of the decisions work out well, and some of them don't.



On or off

You might have heard that computers are based on the binary number system. They can only cope with two extremes: on or off. In the analogy of our maze: left or right.

That switch is called a 'bit'. It is the most fundamental building block of a computer.

So far you have met `int` and `str` types. Now meet your third data type: the **boolean**. Booleans can only be on or off, nothing else. In Python on is called `True` and off is called `False` .

Booleans can help us with many things. They can answer questions like: "*is this true*"? Or "*is this false*"?

A typical question to ask is whether a number is *bigger* or smaller than another number.

In Python we can write this with `>` . So to test whether 7 is bigger than 6, write `7 > 6` below:

In [2]:

```
7 > 6
```

Out[2]:

```
False
```

That was `True` ! Of course. Which of these things are `True` , and which of them are `False` ?

In [ ]:

```
7 > 8
```

In [ ]:

```
2 < 3
```

In [3]:

```
-1 > 100
```

Out[3]:

```
False
```

Decisions, decisions

Booleans can help us to take decisions, because we can learn whether they are `True` or `False` . Think about the maze: a good binary question to ask would be "is this a dead end"?

A computer will blindly follow your instructions. At all times. And you have one tool to help it decide: **boolean**s.

Python has been clever and mimiced the english language. You can write such a decision like so:

```
if is_dead_end:
    'Go right'
else:
    'Go left'
```

Notice that we introduced the variable `is_dead_end` . In the following code the variable is replaced with an actual boolean value. Can you see why that code doesn't make sense?

In [15]:

```python
print(6==5)
if True:
    print('Go right')
else:
    print('Go left')
```

```
False
Go right
```

Write this into Mu and see what happens.

Congratulations! You just built a brain-damaged robot: it will never ever ever turn left. Why? Because `True` is always `True` . It will never get to the `else` clause and go left. The magic in the decisions arrive from the fact that we **don't** know whether our condition is `True` or `False` . That's why we need a variable.

Can you make the code above print `Go left` by only changing the first line?

Invisible strings

Notice that the `print` statements inside the `if` statement is indented: it is not on the same vertical line as the `if` and `else` words. Because the `print` s in a sense are following the outer expressions (`if` and `else`), they are said to be **nested**. Python is really strict about this. Let's see what happens if we forget it:

In [10]:

```python
moment_of_truth = False
if moment_of_truth:
    print('Moment of Truth!')
```

```
  File "<ipython-input-10-6c3dd4a5e8d1>", line 3
    print('Moment of Truth!')
        ^
IndentationError: expected an indented block
```

Crash! This is totally on purpose. Why? Because it is much more readable. Consider these two options:

In [11]:

```
if moment_of_truth:
print('Moment of Truth!')
else:
print('Let it Shine')
```

```
  File "<ipython-input-11-5c2724bb54b4>", line 2
    print('Moment of Truth!')
        ^
IndentationError: expected an indented block
```

In [12]:

```
if moment_of_truth:
    print('Moment of Truth!')
else:
    print('Let it Shine')
```

```
Let it Shine
```

I hope you'll agree that indentation is a good thing. The second option is much clearer.

What actually happens behind the scene is that Python *insists* that you prepend your `print` statement with 4 spaces. In a Python string, that would look like this:

In [ ]:

```
'    '
```

And space (`' '`) is actually a character just like `'a'` is a character.

You can make a string entirely of spaces, just like you can make a string of `'a'` s. So we can work with it like we did before.

Before you execute the code below, try to explain it to your neighbour and fully understand what you expect it to do:

In [13]:

```
space = input('Space between us: ')
spaces = ' ' * int(space)
print('We are ' + spaces + ' apart')
```

```
Space between us: 10
We are            apart
```

Magic (and invisible) strings

Spaces are not the only invisible character. Can you guess what this character does?

In [18]:

```
print('Python, \ngive me a break! \nmore')
```

```
Python,
give me a break!
more
```

The `'\n'` stands for "new line". It simply inserts a new line. Try to modify the program below to use `'\n'` instead of a space `' '`.

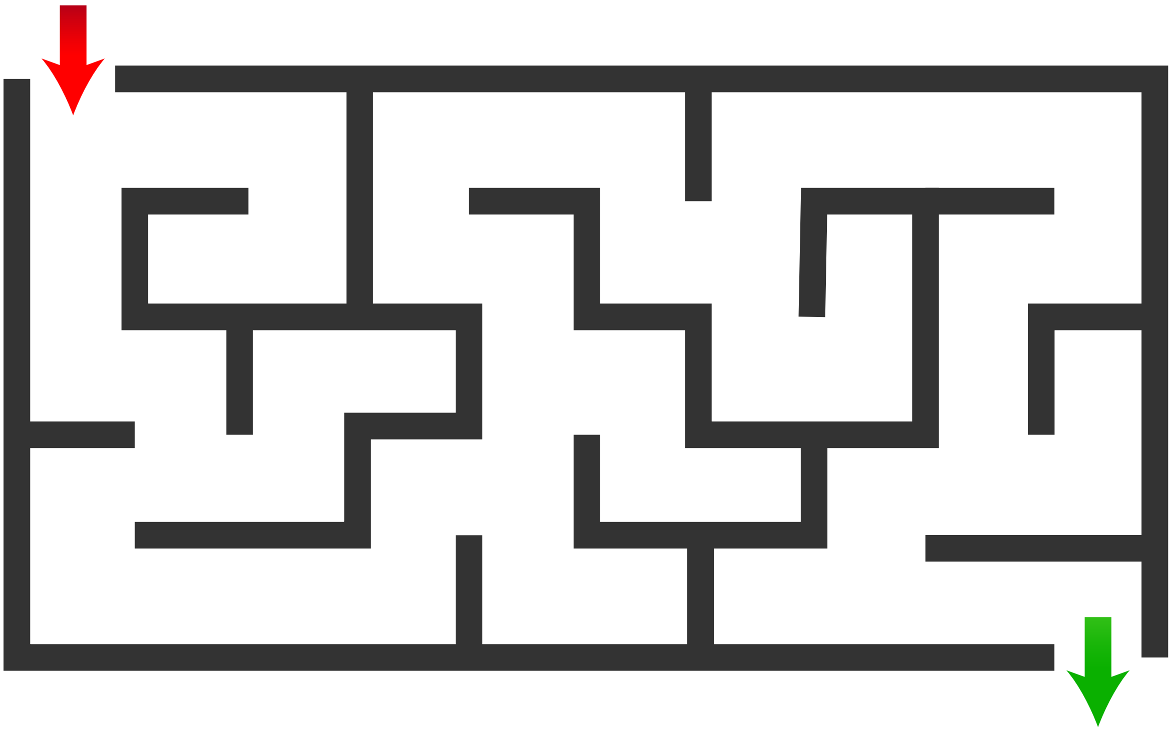
In [ ]:

```
space = input('Space between us: ')
print('We are ' + ' ' * int(space) + ' apart')
```

Machines!

Machines can't really take any kinds of decision by themselves. The idea of computers as evil machines, is far from reality.

A better picture of a computer is like a maze. When we tell it to, it will start to execute a program (the maze) and then it starts taking a number of decisions. Some of the decisions work out well, and some of them don't.



In [ ]:

```

```

String Equality

So far we have seen how to test for boolean conditions with numbers. For instance: is `7 > 6`? We can also do this with text strings.

For example, you can ask whether strings are the same. Python writes that with two equal symbols: `==` . Try to run the following code:

In [19]:

```
'a' == 'a'
```

Out[19]:

```
True
```

In [20]:

```
'a' == 'b'
```

Out[20]:

```
False
```

In [21]:

```
'hullu bullu, lotte hvor er du henne?' == 'hullu bullu lotte hvor er du henne
```

Out[21]:

```
False
```

Which one of these statements are `True` ? Try to guess the solution before you execute the code.

In [22]:

```
'FCK' == 'fck'
```

Out[22]:

```
False
```

In [23]:

```
'Faxe Kondi' == 'Apple Juice'
```

Out[23]:

```
False
```

In the same way numbers are ordered, so 7 really **is** bigger than 6, strings are ordered lexicographically. That is perhaps not of too much practical relevance for now, but it allows for simple and funny insights:

In [24]:

```
'Faxe Kondi' > 'Apple Juice'
```

Out[24]:

```
True
```

You don't love cookie dough enough!

Your previous cookie dough program probably looked something like this:

```
data = input('How much do you like cookie dough?')
data = int(data)
print('You ' + 'really ' * data + 'like cookie dough')
```

But, what happens if you give the program a number that is 0 or less? Try it out for yourself.

In that program it's possible to output `'You like cookie dough'` if `0` is given as input (`0 * 'really' == ''`). `I think you'll agree that we need at least one 'really'` in there.

Let's fix this. Your job is to tell off people that are putting in numbers that are less than 1. If they do so, you should print out the string `'You don't like cookie dough enough!'` . If the number is above 0, you should print out the same statement as before.

Hint: Use the `if` notation from above.

In [ ]:

```

```