Exercise 1: ordering pancakes at the baker

Snap! runtime verification assignment - 🇬🇧

This assignment is about a pancake baker selling pancakes. His pancakes are very good, but unfortunately the baker is not very good at calculating. In Task 1 we will find out what the baker is doing wrong. Then, we introduce a block for writing robust scripts: the **assert** block. In Task 2 we will show you how to use the **assert** block in the code. In Task 3 we will teach the baker how to be good at calculating.

This exercise assumes that you know Snap! or Scratch already. If this is not the case, ask us for help, or have a look at the Scratch exercises. They introduce you to the basics of Snap! and Scratch.

Before we begin: meet the baker

However, before we start, we will first briefly explain to you how the baker works! This is the pancake baker. He sells pancakes for 3 euros per pancake. There is one important detail: just below his pancake stand, there is a number in a box. In the photo below it is “amount of batter 10”. This number indicates how much batter the pancake baker has left for baking pancakes.

3€
P.P
amount of batter 10

Task 1: Ordering pancakes

Now that you know how the baker works, run the script a few times so you know what it does. Once you're ready, try the following:

- Try ordering 5 pancakes. What happens? Is this what you expect?
- Try ordering 15 pancakes. What happens? Is this what you expect?
- Finally, try ordering -5 pancakes. What happens? Does this make sense?

Answer

The baker does fine when you ask him for 5 pancakes. But the baker does not know their limits! The baker is fine with baking 15 pancakes, even though the baker has only enough batter for 10. The baker is also fine with making -5 pancakes, even though this is impossible.

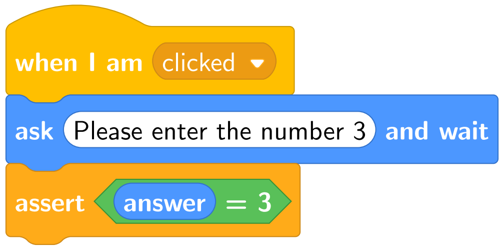
New block: assert

For the next exercise, you need to learn about a new block. This block is called the "**assert**" block. You can find it in the "Control" part of Snap!.

You can think of this as a block for finding problems. When Snap! reaches this block, Snap! considers the block within the assert block. If the result evaluates to true, Snap! continues to the next block. If the result evaluates to false, Snap! shows an error message and does not continue to the next block. When the result is false and an error message appears, we say that "the assert is triggered". In a way, if you can find a way to write down your problem in a block, the assert block helps you find that problem as soon as it occurs. In other words, when the assert is triggered, you know something went **wrong!**

Assert block example

Let's create a small example to understand it better. We'll make a small Snap! script that first asks you for a value, and then uses the **assert** block to make sure the value is equal to 3. In your Snap! project, create the following blocks:



```
when I am clicked
ask Please enter the number 3 and wait
assert answer = 3
```

Then, click the "**when I am clicked**" block.

- What happens if you enter the number 5? Does that make sense?
- What happens if you enter the number 3? Is that what you expect?

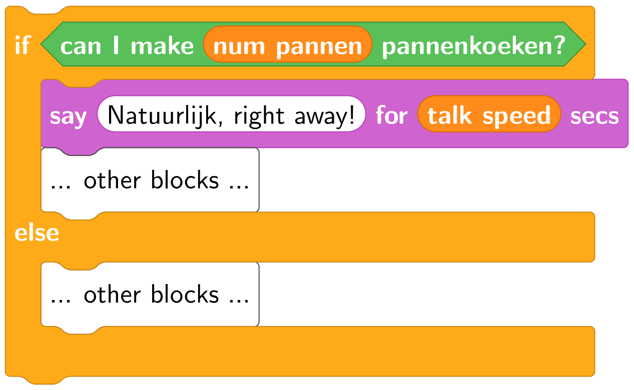
Answer

If you enter 5, Snap! tells you that something went wrong. This makes sense, because 5 is not equal to 3! If you enter 3, nothing happens. This makes sense, because 3 *is* equal to 3.

Task 2: Detecting problems

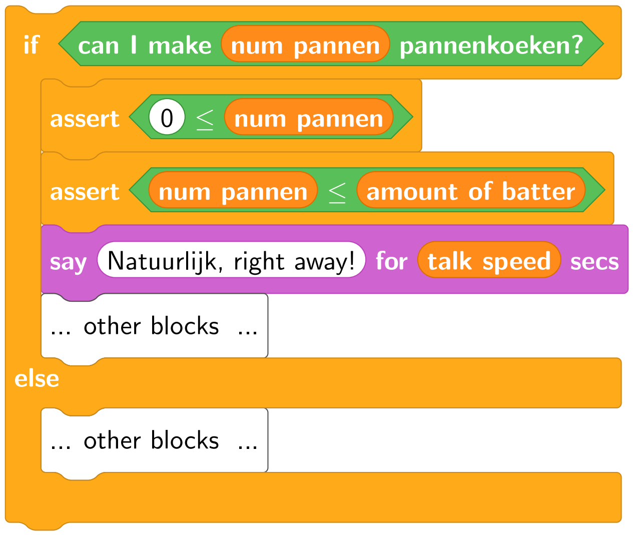
Now that you know about the assert block, we will use it to detect when the customer asks for a number of pancakes that the baker cannot bake.

In the script of the baker, there is the following sequence of blocks:



```
if can I make num pannen pannenkoeken?
    say Natuurlijk, right away! for talk speed secs
    ... other blocks ...
else
    ... other blocks ...
```

We will now add asserts to detect whenever the baker will bake a negative or too large number of pancakes. Modify this block to look as follows:



```
if <can I make (num pannen) pannenkoeken?>
    assert <(0) ≤ (num pannen)>
    assert <(num pannen) ≤ (amount of batter)>
    say (Natuurlijk, right away!) for (talk speed) secs
    ... other blocks ...
else
    ... other blocks ...
```

Now, lets try ordering pancakes again!

- Try ordering 5 pancakes. What happens? Is this what you expect?
- Try ordering 15 pancakes. What happens? Is this what you expect?
- Finally, try ordering -5 pancakes. What happens? Does this make sense?

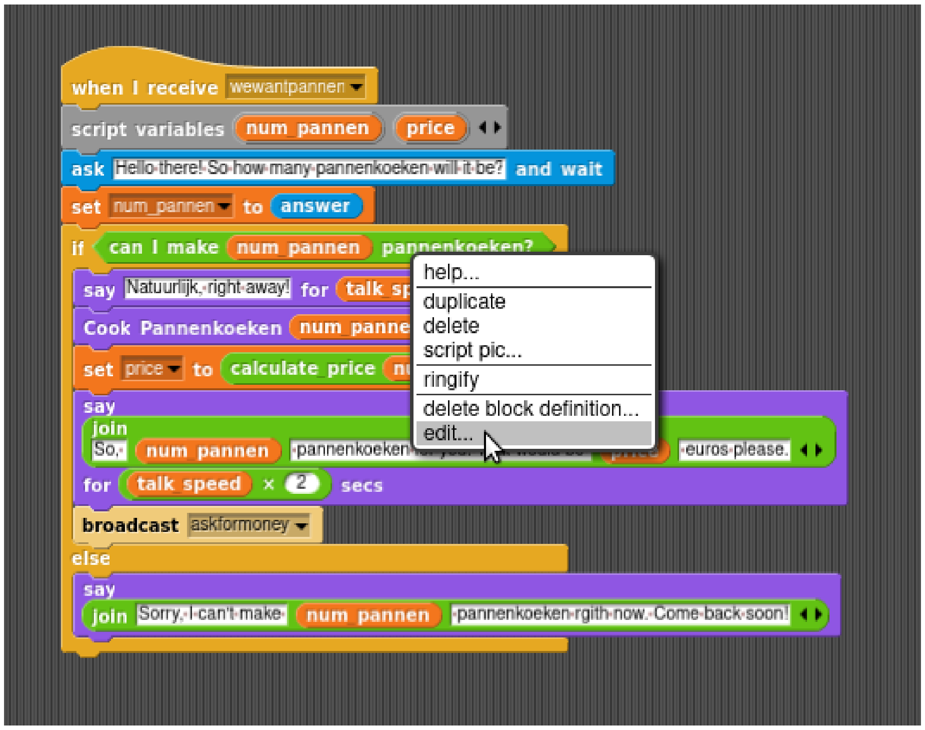
Answer

If you added the right asserts, ordering 5 pancakes should be no problem. When ordering -5 or 15 pancakes, Snap! should now give you an error that an **assert** was triggered.

The current script behaviour might seem worse than what the baker did before: maybe you don't mind that the baker can have infinite batter! For this exercise however, we have decided that it is important not to let this happen. Therefore, we want to know when something goes wrong!

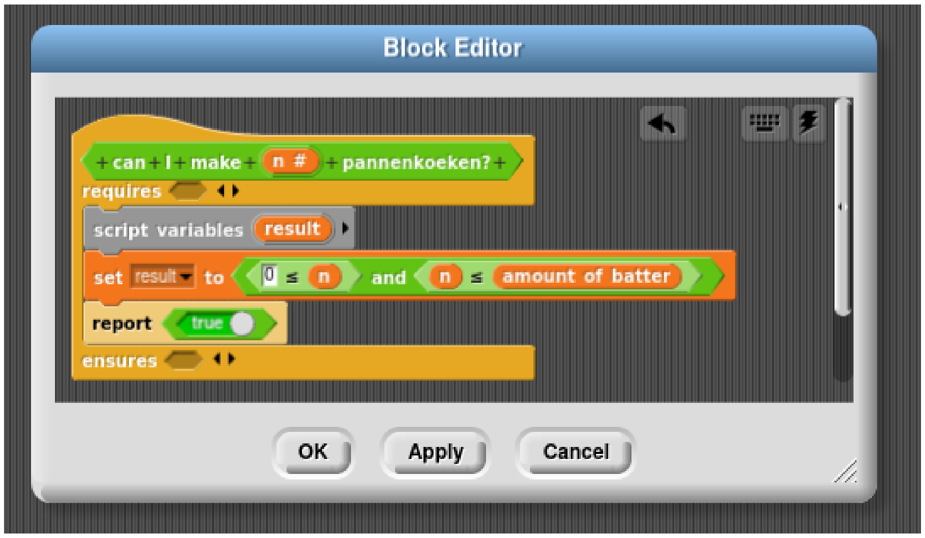
Task 3: Teaching the baker to calculate

If the baker knew how to calculate properly, the **assert** blocks would never be triggered. So lets teach the baker how to calculate! We will modify the **can I make ( ) pannenkoeken?** block. To do this, right click on the block and click "edit...":



```
when I receive wewantpannen
script variables num_pannen price
ask Hello there! So how many pannenkoeken will it be? and wait
set num_pannen to answer
if can I make num_pannen pannenkoeken?
    say Natuurlijk, right away! for talk sp
    Cook Pannenkoeken num_panne
    set price to calculate_price n
    say
        join So, num pannen pannenkoeken                    euros please.
    for talk speed × 2 secs
    broadcast askformoney
else
    say
        join Sorry, I can't make num pannen pannenkoeken rgith now. Come back soon!
```

The following window should appear:



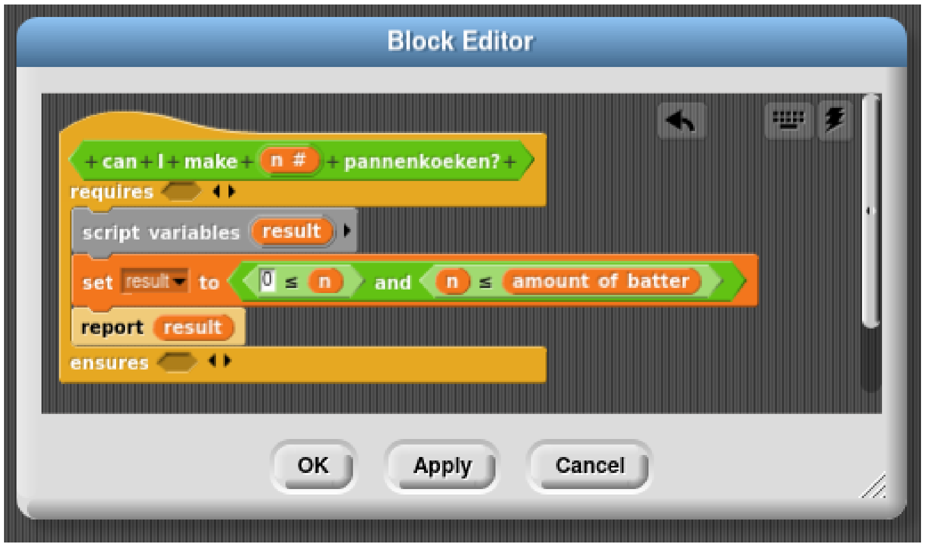
```
can I make n # pannenkoeken?
requires
    script variables result
    set result to 0 ≤ n and n ≤ amount of batter
    report true
ensures
```

Oh no! It looks like the baker was checking if there is enough batter, but reporting the wrong result. Can you explain why?

Answer

Inside can I make ◯ pannenkoeken?, result is set to true ● if n is between 0 and amount of batter. However, result is not actually used after that! Instead, true ● is reported, meaning that the result is not actually used. Hence the wrong result is reported.

To fix this, report the result instead of true ●. To do this, drag result from the **script variables** to the report ◯ block. Your final block should look like this:



```
+can+I+make+ n # +pannenkoeken?+
requires
script variables result
set result to (0 ≤ n) and (n ≤ amount of batter)
report result
ensures
```

Click "**OK**", and try all the inputs again!

- Try ordering 5 pancakes. What happens? Is this what you expect?
- Try ordering 15 pancakes. What happens? Is this what you expect?
- Finally, try ordering -5 pancakes. What happens? Does this make sense?

Answer

When ordering 5 pancakes, the baker bakes 5 pancakes. When ordering 15 or -5 pancakes, the baker should tell you that it cannot do this.

You taught the baker how to calculate, and the asserts are no longer triggered. Good job!

Summary

By doing Task 1, 2 and 3, you have learned to find problems by looking at a Snap! script, executing it, and formulating the problem as assert blocks that detect it. This is actually an example of good "**software engineering practice**": whenever you make an assumption about the behaviour of your program, it is good practice to make this assumption explicit as an assert. This serves not only as documentation for someone else reading your code, but also make sure that you get an error if this assumption does not hold as expected.

To relate this to the pancake baker: if we ever change the code, and maybe introduce a bug, the asserts will be triggered whenever the number of pancakes ordered goes below 0 or above the amount of available batter. This gives us more confidence in the behaviour of the baker.