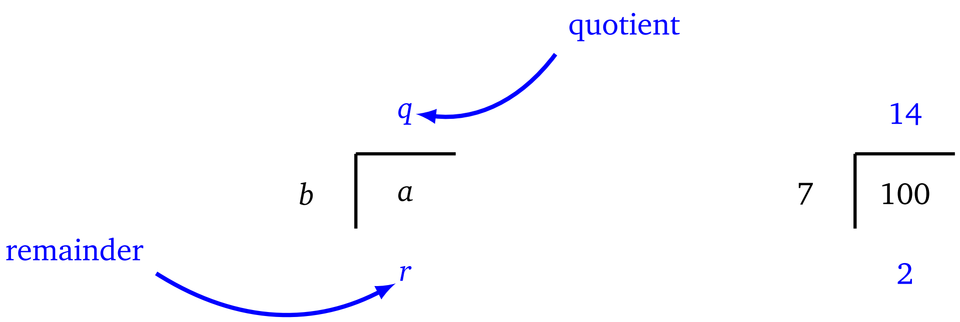
# Arithmetic – While loop – I

*The activities in this sheet focus on arithmetic: long division, prime numbers . . . This is an opportunity to use the "while" loop intensively.*

**Lesson 1** (Arithmetic).
Let us recall what Euclidean division is. Here is the division of $a$ by $b$, $a$ is a positive integer, $b$ is a strictly positive integer (with the example of 100 divided by 7):




We have the two fundamental properties that define $q$ and $r$:

$$a = b \times q + r \qquad \text{and} \qquad 0 \leqslant r < b$$

For example, for the division of $a = 100$ by $b = 7$: we have the quotient $q = 14$ and the remainder $r = 2$ that verify $a = b \times q + r$ because $100 = 7 \times 14 + 2$ and also $r < b$ because $2 < 7$.

With `Python`:

- `a // b` returns the quotient,
- `a % b` returns the remainder.

It is easy to check that:

$b$ is a divisor of $a$ if and only if $r = 0$.

**Activity 1** (Quotient, remainder, divisibility).

*Goal: use the remainder to find out if one integer divides another.*

1. Program a function named `quotient_remainder(a,b)` that does the following tasks for two integers $a \geqslant 0$ and $b > 0$:
   - It displays the quotient $q$ of the Euclidean division of $a$ per $b$,
   - it displays the remainder $r$ of this division,

- it displays `True` if the remainder $r$ is positive or zero and strictly less than $b$, and `False` otherwise,
- it displays `True` if you have equality $a = bq + r$, and `False` if not.

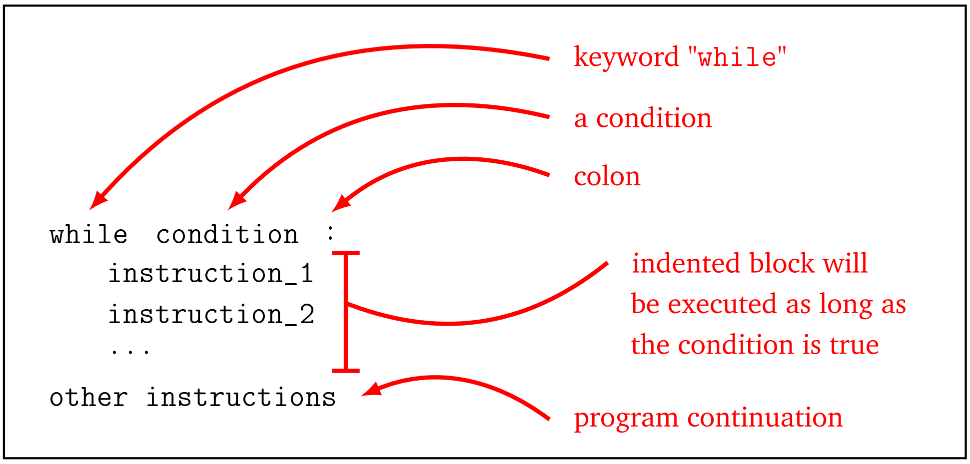
Here is an example of what the call should display for `quotient_remainder(100,7)`:

```
Division of a = 100 by b = 7
The quotient is q = 14
The remainder is r = 2
Check remainder: 0 <= r < b? True
Check equality: a = bq + r? True
```

*Note.* You have to check without cheating that we have $0 \leqslant r < b$ and $a = bq + r$, but of course it must always be true!

2. Program a function called `is_even(n)` that tests if the integer $n$ is even or not. The function should return `True` or `False`.
*Hints.*
   - First possibility: calculate `n % 2`.
   - Second possibility: calculate `n % 10` (which returns the digit of units).
   - The smartest people will be able to write the function with only two lines (one for `def...` and the other for `return...`).
3. Program a function called `is_divisible(a,b)` that tests if $b$ divides $a$. The function should return `True` or `False`.

**Lesson 2** ("while" loop).
The "while" loop executes instructions as long as a condition is true. As soon as the condition becomes false, it proceeds to the next instructions.



**Example.**
Here is a program that displays the countdown $10, 9, 8, \ldots 3, 2, 1, 0$. As long as the condition $n \geqslant 0$ is true, we reduce $n$ by 1. The last value displayed is $n = 0$, because then $n = -1$ and the condition "$n \geqslant 0$" becomes false so the loop stops.

```
n = 10
while n >= 0:
    print(n)
    n = n - 1
```

This is summarized in the form of a table:

Input: $n = 10$

| $n$ | "$n \geqslant 0$" ? | new value of $n$ |
|---|---|---|
| 10 | yes | 9 |
| 9 | yes | 8 |
| … | … | … |
| 1 | yes | 0 |
| 0 | yes | $-1$ |
| $-1$ | no | |

Display: $10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0$

**Example.**

This piece of code looks for the first power of 2 greater than a given integer $n$. The loop prints the values 2, 4, 8, 16,… It stops as soon as the power of 2 is higher or equal to $n$, so this program displays 128.

```
n = 100
p = 1
while p < n:
    p = 2 * p
print(p)
```

Inputs: $n = 100$, $p = 1$

| $p$ | "$p < n$" ? | new value of $p$ |
|---|---|---|
| 1 | yes | 2 |
| 2 | yes | 4 |
| 4 | yes | 8 |
| 8 | yes | 16 |
| 16 | yes | 32 |
| 32 | yes | 64 |
| 64 | yes | 128 |
| 128 | no | |

Display: 128

**Example.**
For this last loop we have already prepared a function called `is_even(n)` which returns `True` if the integer $n$ is even and `False` otherwise. The loop does this: as long as the integer $n$ is even, $n$ becomes $n/2$. This amounts to removing all factors 2 from the integer $n$. As $n = 56 = 2 \times 2 \times 2 \times 7$, this program displays 7.

```
n = 56
while is_even(n) == True:
    n = n // 2
print(n)
```

Input: $n = 56$

| $n$ | "is $n$ even" ? | new value of $n$ |
|---|---|---|
| 56 | yes | 28 |
| 28 | yes | 14 |
| 14 | yes | 7 |
| 7 | no | |

Display: 7

For the latter example, it is much more natural to start the loop with

```
while is_even(n):
```

Indeed `is_even(n)` is already a value "True" or "False". Therefore we're getting closer to the English sentence "while $n$ is even..."

**Operation "+=".** To increment a number you can use these two methods:

`nb = nb + 1` or `nb += 1`

The second writing is shorter but makes the program less readable.

**Activity 2** (Prime numbers).

*Goal: test if an integer is (or not) a prime number.*

1. **Smallest divisor.**
   Program a function called `smallest_divisor(n)` that returns, the smallest divisor $d \geqslant 2$ of the integer $n \geqslant 2$.
   For example `smallest_divisor(91)` returns 7, because $91 = 7 \times 13$.

   **Method.**
   - We remind you that $d$ divides $n$ if and only if `n % d` is equal to 0.
   - It is a bad idea to use a loop "for $d$ ranging from 2 to $n$", since, if for example we know that 7 is a divisor of 91 it is useless to test if $8, 9, 10\ldots$ are also divisors because we have already found a smaller one.
   - A good idea is to use a "while" loop! The principle is: "as long as I haven't got my divisor, I should keep looking for". (And so, as soon as I find it, I stop looking.)
   - In practice here are the main lines:
     - Begin with $d = 2$.
     - As long as $d$ does not divide $n$ move on to the next candidate ($d$ becomes $d + 1$).
     - At the end $d$ is the smallest divisor of $n$ (in the worst case $d = n$).
2. **Prime numbers (1).**

Slightly modify your `smallest_divisor(n)` function to write your first prime function `is_prime_1(n)` which returns "True" if $n$ is a prime number and "False" otherwise.
For example `is_prime_1(13)` returns `True`, `is_prime_1(14)` returns `False`.

3. **Fermat numbers.**
   Pierre de Fermat (∼1605–1665) thought that all integers of the form $F_n = 2^{(2^n)} + 1$ were prime numbers. Indeed $F_0 = 3$, $F_1 = 5$ and $F_2 = 17$ are prime numbers. If he had known `Python` he would probably have changed his mind! Find the smallest integer $F_n$ which is not prime.
   *Hint.* With `Python` $b^c$ is written `b ** c` and therefore $a^{(b^c)}$ is written `a ** (b ** c)`.

   *We will improve our function which tests if a number is prime or not, it will allow us to test lots of numbers or very large numbers more quickly.*

4. **Prime numbers (2).**
   Enhance your previous function to become `is_prime_2(n)`. It should not test all the divisors $d$ from 2 to $n$, but only up to $\sqrt{n}$.
   *Explanations.*
   - For example, to test if 101 is a prime number, just see if it divisible by $2, 3, \ldots, 10$. It is faster!
   - This improvement is due to the following proposal: if an integer is not prime then it admits a divisor $d$ that verifies $2 \leqslant d \leqslant \sqrt{n}$.
   - Instead of testing if $d \leqslant \sqrt{n}$, it is easier to test if $d^2 \leqslant n$.

5. **Prime numbers (3).**
   Improve your function to become `is_prime_3(n)` using the following idea. We test if $n$ is divisible by $d = 2$, but from $d = 3$, we just test the odd divisors (we test $d$, then $d + 2 \ldots$).
   - For example to test if $n = 419$ is a prime number, we first test if $n$ is divisible by $d = 2$, then $d = 3$ and then $d = 5$, $d = 7 \ldots$
   - This allows you to do about half less tests!
   - Explanations: if an even number $d$ divides $n$, then we already know that 2 divides $n$.

6. **Calculation time.**
   Compare the calculation times of your different functions `is_prime()` by repeating the call `is_prime(97)`, for example, a million times. See the course below for more information on how to do this.

**Lesson 3** (Calculation time)**.**
There are two ways to make programs run faster: a good way and a bad way. The bad way is to buy a more powerful computer. The good method is to find a more efficient algorithm!
With `Python`, it is easy to measure the execution time of a function in order to compare it with the execution time of another. Just use the module `timeit`.
Here is an example: we measure the computation time of two functions that have the same purpose, test if an integer $n$ is divisible by 7.

```
# First function (not very clever)
def my_function_1(n):
    divis = False
    for k in range(n):
        if k*7 == n:
            divis = True
    return divis
```

```
# Second function (faster)
def my_function_2(n):
    if n % 7 == 0:
        return True
    else:
        return False

# Measurement of execution times
import timeit

print(timeit.timeit("my_function_1(1000)",
    setup="from __main__ import my_function_1",
    number=100000))
print(timeit.timeit("my_function_2(1000)",
    setup="from __main__ import my_function_2",
    number=100000))
```

**Results.**

The result depends on the computer, but allows the comparison of the execution times of the two functions.

- The measurement for the first function (called 100 000 times) returns 5 seconds. The algorithm is not very clever. We're testing if $7 \times 1 = n$, then test $7 \times 2 = n$, $7 \times 3 = n \ldots$
- The measurement for the second function returns 0.01 second! We test if the remainder of $n$ divided by 7 is 0. The second method is therefore 500 times faster than the first.

**Explanations.**

- The module is named `timeit`.
- The function `timeit.timeit()` returns the execution time in seconds. The function takes the following parameters:
  - a string for the call of the function to be tested (here we ask if 1000 is divisible by 7),
  - an argument `setup="..."` which indicates where to find this function,
  - the number of times you have to repeat the call to the function (here `number=100000`).
- The number of repetitions must be large enough to avoid uncertainties.

**Activity 3** (More prime numbers).

*Goal: program more "while" loops and study different kinds of prime numbers using your* `is_prime()` *function.*

1. Write a `prime_after(n)` function that returns the first prime number $p$ greater than or equal to $n$.
   For example, the first prime number after $n = 60$ is $p = 61$. What is the first prime number after $n = 100\,000$?
2. Two prime numbers $p$ and $p+2$ are called ***twin prime numbers***. Write a `twin_prime_after(n)` function that returns the first pair $p$, $p+2$ of twin prime numbers, with $p \geqslant n$.
   For example, the first pair of twin primes after $n = 60$ is $p = 71$ and $p + 2 = 73$. What is the first pair of twin primes after $n = 100\,000$?
3. An integer $p$ is a ***Germain prime number*** if $p$ and $2p + 1$ are prime numbers. Write a

`germain_after(n)` function that returns the pair $p$, $2p+1$ where $p$ is the first Germain prime number $p \geqslant n$.
For example, the first Germain prime number after $n = 60$ is $p = 83$, with $2p + 1 = 167$. What is the first Germain prime number after $n = 100\,000$?