General

Formatted text: string interpolation

When you need a controlled way to compose a string as a mix of fixed and variable fragments, it might be handy to make use of string interpolation. An **interpolated string** is a regular string that is prefixed with the letter `f` (in front of the opening single or double quote). As a result, interpolated strings are also called **f-strings**.

An f-string serves as a kind of template, with each variable fragment indicated by a pair of curly braces (`{}`). In between these curly braces you place an expression whose value will fill up the position of the variable fragment in the resulting string.

For example, in the following code fragment we define two variables `number1` and `number2` whose sum we want to output. We use string interpolation to output formatted text that contains the two individual terms and the result of adding the two terms.

```
>>> number1 = 2
>>> number2 = 3
>>> print(f'The sum of {number1} and {number2} is {number1 + number2}.')
The sum of 2 and 3 is 5.
```

A pair of curly braces in an interpolated string is called a **placeholder**. Inside such a placeholder you cannot only put an expression, but after a colon you can also specify how the value of that expression must be formatted (read: how it needs to be converted into a fixed string). More details about the different ways to specify this formatting can be found in The Python Standard Library.

How does Dodona check floating point numbers

If you have to output a *floating point* number for a given assignment, without an explicit indication about the exact number of decimal digits that has to be displayed on the output (without rounding or truncating), Dodona will check by default that the number is accurate up to six decimal digits. As a result, it does not really matter how many digits are shown on the output.

Best laid plans

Output floats with a fixed number of decimal digits (rounded)

By default the built-in function `print` formats floating point numbers using a large number of decimal digits. However, sometimes you will want to print floating point numbers with a fixed number of decimal digits. You could try to use the built-in function `round` to achieve this, as it allows rouning of numbers up to a given number of decimal digits.

```
>>> print(1 / 3)
0.3333333333333333
>>> print(round(1 / 3, 2))
0.33
```

The problem with this solution is that rounding errors due to the internal representation of floating point numbers, may generate numbers that are not printed with the desired number of decimal digits.

A better solution makes use of string interpolation to specify the number of decimal digits when formatting floating point numbers as text. Inside a pair of curly braces that represents a placeholder in the template string, you may specify how the value that fills up the placeholder must be formatted. This is done by placing a so-called *format specifier* in between the curly braces. The format specifier itself is preceded by a colon (`:`).

To format a value as a floating point number with a fixed number of decimal digits, you can use the format specifier `:.nf`. Here, the letter `f` indicates that the value must be formatted as a floating point number, and

the number n indicates the number of decimal digits. The following code shows, for example, how a number can be formatted as a floating point number, rounded up to two decimal digits.

```
>>> print(f'{1 / 3:.2f}')
0.33
```

We refer to The Python Standard Library for more details about the use of *format specifiers*.

Multiplication: operator *

Do not forget to use the operator * if you want to multiply two numbers with each other. In mathematics it is commonplace to write multiplication without any operator. In this case, the notation xy is used for example to indicate the product of x and y. Python forces you to write the multiplication explicitly, by using the operator *.

```
>>> x = 6
>>> y = 7
>>> x * y
42
>>> xy
Traceback (most recent call last):
NameError: name 'xy' is not defined
```

The pudding guy

Floating point division versus integer division

Python makes a clear distinction between floating point division (indicated by the operator `/`) and integer division (indicated by the operator `//`). Floating point division always results in a `float`. However, with integer division, the data type of the result depends on the data type of the operandi. If both operandi are integers, the result is an integer as well. If one or two of the operandi are `floats`, the result is itself a `float`.

```
>>> x = 8
>>> y = 3
>>> z = 4
>>> x / y            # floating point division of two integers
2.6666666666666665
>>> x // y           # integer division of two integers
2
>>> float(x) // y    # integer division of a float and an integer
2.0
>>> x / z            # floating point division of two integers
2.0
>>> x // z           # integer division of two integers
2
```

Python decides which kind of division to use solely based on the operator that is being used. The choice between floating point division or integer division is not influenced by the data types of the operandi.

```
>>> x = 7.3
>>> y = 2
>>> x // y
3.0
>>> y // x
0.0
>>> x / y
3.65
```

Human development index

Extra mathematical functionality: the `math` module

It is an explicit design choice to keep the Python programming language as small as possible. However, there are mechanisms built into the language to extend the language with new functionality. When Python is installed, a selection of these modules are shipped along. These modules are referred to as The Python Standard Library.

The `math` module is one of these modules from the The Python Standard Library. As you might derive from its name, the `math` module adds some mathematical functionality to Python. Before you can start using this functionality, however, you must first import the module. There are two ways in which this can be done.

The first way imports the module as a whole. After this has been done, you must prefix the names of variables, functions or classes that are defined in the module with the name of the module and a dot if you want to use them in your Python code.

```
>>> import math
>>> math.sqrt(16)          # square root
4.0
>>> math.log(100)          # natural logarithm
4.605170185988092
>>> math.log(100, 10)      # log10
2.0
>>> math.pi                # accurate value of pi
3.141592653589793
```

The second way only imports some specific names of variables, functions or classes in your Python code. After this has been done, you can directly use these names without prefixing them.

```
>>> from math import sqrt, log, pi
>>> sqrt(16)               # square root
4.0
>>> log(100)               # natural logarithm
4.605170185988092
>>> log(100, 10)           # log10
2.0
>>> pi                     # naccurate value of pi
3.141592653589793
```

We refer to The Python Standard Library for a complete overview of the variables and functions defined in the `math` module.

Square root

The square root of a number can be computed using the `sqrt` function from the `math` module.

```
>>> import math
>>> math.sqrt(121)
11.0
>>> math.sqrt(1234)
35.12833614050059
```

Because $\sqrt{x} = x^{1/2}$ the power operator (`**`) can be used as well to compute the square root.

```
>>> 121 ** (1 / 2)
11.0
>>> 1234 ** 0.5
35.12833614050059
```

The cubic, fourth, ... root can be calculated as follows:

```
>>> 27 ** (1 / 3)
3.0
>>> 22 ** (1 / 4)
2.1657367706679937
```

Specific information

As a debugging aid, we provide you with some intermediate values that need to be computed before you can generate the output corresponding to the given sample input:

$$\begin{aligned} \text{LEI} &= 0.9718780096308187 \\ \text{EI} &= 0.846147794929596 \\ \text{MYSI} &= 0.8233902000000001 \\ \text{EYSI} &= 0.7864077669902911 \\ \text{II} &= 0.8562070881051073 \end{aligned}$$

The stopped clock

Remainder after integer division: the modulo operator (%)

In Python you can use the modulo operator (`%`) to determine the remainder after integer division. If both operandi are integers, the result is itself an integer. As soon as one of the operandi is a `float`, the result will be a 'float.

```
>>> 83 % 10
3
>>> 83.0 % 10
3.0
>>> 83 % 10.0
3.0
>>> 83.0 % 10.0
3.0
```

Specific information

For this assignment it's a good idea to convert any time indicated by a 24-hour clock into the number of minutes that have elapsed since midnight at that point in time. If the clock indicates $h : m$, then the number of elapsed minutes since midnight equals

$$60 \times h + m$$

Say that the number of elapsed minutes since midnight equals t, then we can easily invert the above procedure to derive the corresponding time $h : m$ as indicated by a 24-hour clock. In Python you can implement this in the following way:

```
>>> elapsed = 868                 # minutes elapsed since midnight
>>> hours = (elapsed // 60) % 24  # hours elapsed since midnight
>>> minutes = elapsed % 60        # minutes elapsed since last hour

>>> hours
14
```

```
>>> minutes
28
```

Please note that we have made use of integer division (`//`) and remainder after integer division (`%`). The use of the modulo operator in determing the number of hours that have elapsed since midnight (`% 24`) makes sure the hours on a 24-hour clock are still correctly computed in case the number of minutes that have elapsed since midnight is longer than a full day.

Using the above method we can now compute the total number of minutes that Andrea has been away from her home, and the total number of minutes she has spent at her friend's house. In doing so, we only need to take into account that the start and end time points of the time interval do not necessarily have to fall in the same day. For example, Andrea can leave home before midnight and only return home after midnight.