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.

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
```

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 mechanism built into the language to extend the language with new functionality. When Python is installed, a selection of these modules are shipped as well. 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.

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
```

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.

Where is the father?

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
```

`TypeError: unsupported operand type(s) for ....`

Double check if you have taken into account that the built-in function `input` always returns an integer. In most cases it may be necessary to convert the result into an object with the appropriate data type. You may use one of the built-in type conversion functions (`int`, `float`, `str`, ...) to achieve this.

```
>>> age = input('How old are you? ')
How old are you? 3
>>> 10 + age
Traceback (most recent call last):
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> 10 + int(age)
13
```

Specific information

This problem can be rewritten as a linear system with two equations in two variables. If we represent the age of the mother (in months) as the variable m and the age of her son as the variable s, we can derive the following two equations from the problem description:

- A mother is a years older than her son:
$$m = s + 12a$$
- b years from now, she will be c times his age:
$$m + 12b = c(s + 12b)$$

Note that we have immediately multiplied the values a and b by 12, because all time intervals are expressed in months and the values a and b are initially given in years. As a result, we have to solve the following system of equations for m and s

$$\begin{cases} m = s + 12a \\ m + 12b = c(s + 12b) \end{cases}$$

We may for example substitute m from the first equation in the second equation so that we obtain the following equality

$$s + 12a + 12b = c(s + 12b)$$

that does not include the variable m. If we solve this equation for s, we obtain

$$\begin{aligned} cs + 12bc &= s + 12a + 12b \\ cs - s &= 12a + 12b - 12bc \\ s(c-1) &= 12(a + b - bc) \\ s &= \frac{12(a + b - bc)}{c - 1} \end{aligned}$$

Now that we have been able to determine the age of the son, we can substitute that value in the first equation to obtain the age of the mother

$$m = s + 12a$$

Note: As an extra tip we advise you not to copy formulas directly from a PDF file. Copying from a PDF file does not use the correct operators when they are pasted into Python source code.

Great-circle navigation

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

By default the built-in function `print` format 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 this. 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 the string method `format` 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 letterf`‘ 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*.

Trigonometric functions from the `math` module

The `math` module from the Python Standard Library defines a couple of **trigonometric functions** such as the sine function (`sin`), the cosine function (`cos`) and the tangent function (`tan`). It's important to pay attention to the fact that these functions expect an angle expressed in radians, and not in degreees. Luckily enough, the `math` module also defines functions to convert an angle expressed in degrees into radians (`radians`) and *vice versa* (`degrees`).

```
>>> import math
>>> angle = 90
>>> radians = math.radians(angle)
>>> radians
1.5707963267948966
>>> radians == math.pi / 2
True
>>> math.cos(radians) # must evaluate to 0, but note the rounding error
6.123233995736766e-17
>>> math.sin(radians)
1.0
```

Cyclometric functions from the `math` module

The `math` module from the Python Standard Library defines a couple of **cyclometric functions** (or inverse trigonometric functions) such as the arcsine function (`asin`), the arccosine function (`acos`) and the arctangent function (`atan` or `atan2`; the main difference between these two is that the function `atan2` takes into account the quadrant in which the point is located). The domain of the arcsine and the arccosine functions is $[-1, 1]$. This means that these function only take *floating point* values from the interval $[-1, 1]$. As a result, you must take care that rouding errors do not cause values outside this domain. The following example illustrates how the detrimental effect of rounding errors can be remedied.

```
>>> import math
>>> value = 1.00001
>>> math.acos(value)                    # compute arccosine
Traceback (most recent call last):
ValueError: math domain error
>>> value = max(-1.0, min(value, 1.0)) # guarantee that value is in interval [-1, 1]
>>> value
1.0
>>> math.acos(value)
0.0
```