Dogarithmic scale

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.

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 diatomist

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*.

Specific information

A circle with radius r has a circumference of length $2\pi r$ and encloses an area of size πr^2.

Clock hands

Add leading zeros using string method `format`

If you want to convert an integer to fixed-length string with leading zeros, you can use the string method `format` and an accompanying *format specifier*. Format specifiers are always put in between the pair of brackets used as a placeholder in the template string. A format specifier always starts with a colon (`:`).

In particular, to convert an integer into a fixed-length string with leading zero, you use the format specifier `:0nd`. In this format specifier, the `0` indicates leading zeros need to be added in case the string would otherwise be shorter than the target length, the letter `d` indicates that the value must be formatted as a digit (number) and the number `n` indicates the target length of the string. If the number you want to convert to a string is longer than the target length, the entire number is converted into a string. In this case, the resulting string will thus be longer than the target length.

```
>>> f'{2}'
'2'
>>> f'{2:02d}'
'02'
>>> f'{34:02d}'
'34'
```

```
>>> f'{567:02d}'
'567'
>>> f'{89:06d}'
'000089'
```

There are other ways to convert a number into a fixed-length string with leading zeros. One option is to use the string method `zfill`. You can also compute the difference between the actual length and the target length, to determine how many leading zeros must be prepended. Or you can use a `while` loop to prepend leading zeros until the target length is reached.

```
>>> target = 3
>>> number = str(2)
>>> number.zfill(target)
'002'
>>> leading_zeros = target - len(number)
>>> leading_zeros
2
>>> '0' * leading_zeros + number
'002'
>>> while len(number) < target:
...     number = '0' + number
...
>>> number
'002'
```

Determine the smallest value

The built-in function `min` can be used to determine the minimum of two values.

```
>>> min(7, 3)
3
>>> min(3.14, 7.45)
3.14
```

The same function can also be used to determine the minimum of multiple values.

```
>>> min(7, 3, 8, 19, 2, 12)
2
>>> min(3.14, 7.45, 17.35, 373.21, 2.34, 98.36)
2.34
```

Absolute value

The built-in function `abs` can be used to compute the absolute value of a number.

```
>>> abs(42)
42
>>> abs(-42)
42
>>> abs(3.14159)
3.14159
>>> abs(-3.14159)
3.14159
```

Specific information

Each position of a clock hand can be expressed as the number of degrees of the angle it makes (clockwise, of course) with the direction of 12 o'clock. For example, at 3 the clock hand makes an angle of 90° with the direction of 12 o'clock.

The angle in between the two clock hands can then be computed as the difference between their positions expressed in degrees.

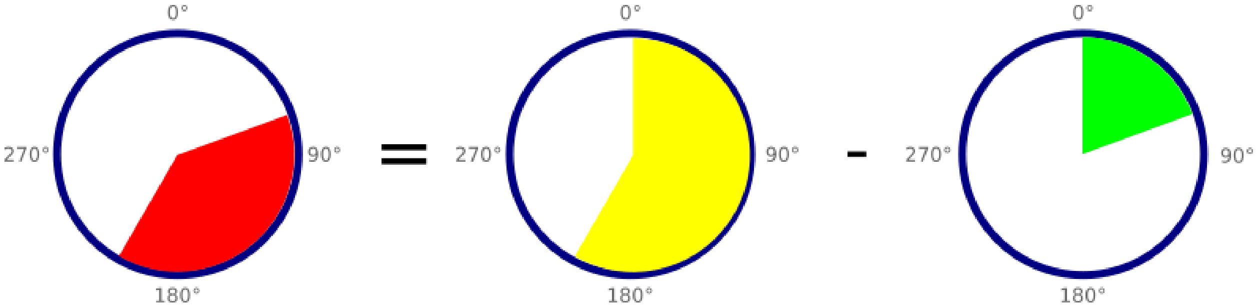

Figure 1: Angles

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
```

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` afloat‘.

```
>>> 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.