Rational numbers in Python (3.x)

floating point numbers, Decimal and Fraction types

Piotr Beling

Uniwersytet Łódzki
(University of Łódź)

2016

Run ipython qtconsole

Today we will work in a python shell.
Please run *ipython qtconsole*:

- Windows with Anaconda: Start → (All) Applications → Anaconda3 → Jupyter QTConsole
- Linux: `ipython3 qtconsole`

Floating point representation – the idea

- In floating point system each number is represented by two integers:
  - *significand* that contains the number's digits,
  - and *exponent* that says where the decimal (or binary, ...) point is placed.
- An example of decimal ($base = 10$) floating point:

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{-4}^{\text{exponent}}}$$

- Binary floating points use $base = 2$.
- `float` type uses binary floating points representation. (Usually it implements IEEE-754 "double precision" standard.)
- Different (than 2 and 10) *base*s are also possible.

Floating point representation – the idea

- In floating point system each number is represented by two integers:
  - *significand* that contains the number's digits,
  - and *exponent* that says where the decimal (or binary, ...) point is placed.
- An example of decimal ($base = 10$) floating point:

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{-4}^{\text{exponent}}}$$

- Binary floating points use $base = 2$.
- `float` type uses binary floating points representation. (Usually it implements IEEE-754 "double precision" standard.)
- Different (than 2 and 10) *base*s are also possible.

Floating point representation – the idea

- In floating point system each number is represented by two integers:
  - *significand* that contains the number's digits,
  - and *exponent* that says where the decimal (or binary, ...) point is placed.
- An example of decimal ($base = 10$) floating point:

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{-4}^{\text{exponent}}}$$

- Binary floating points use $base = 2$.
- `float` type uses binary floating points representation. (Usually it implements IEEE-754 "double precision" standard.)
- Different (than 2 and 10) *base*s are also possible.

Floating point representation – the idea

- In floating point system each number is represented by two integers:
  - *significand* that contains the number's digits,
  - and *exponent* that says where the decimal (or binary, ...) point is placed.
- An example of decimal ($base = 10$) floating point:

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{-4}^{\text{exponent}}}$$

- Binary floating points use $base = 2$.
- `float` type uses binary floating points representation. (Usually it implements IEEE-754 "double precision" standard.)
- Different (than 2 and 10) *base*s are also possible.

Floating point representation – the idea

- In floating point system each number is represented by two integers:
  - *significand* that contains the number's digits,
  - and *exponent* that says where the decimal (or binary, ...) point is placed.
- An example of decimal ($base = 10$) floating point:

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{-4}^{\text{exponent}}}$$

- Binary floating points use $base = 2$.
- `float` type uses binary floating points representation. (Usually it implements IEEE-754 "double precision" standard.)
- Different (than 2 and 10) *base*s are also possible.

Floating point representation – pros and cons

Generally, due to limited memory, no type can represent all numbers accurately.

Pros: A floating-point system can be used to represent numbers of wildly different orders of magnitude.
For example the same data type can store the distance between galaxies and the diameter of an atomic nucleus.

Cons: Many rational numbers cannot be exactly represented with finite memory. For instance:

- $1/3 = 0.333\ldots$ (and also e.g. 1/30, 10/3, ...) does not have an exact finite decimal floating point representation;
- $2/10 = 0.00110011\ldots_2$ (and also e.g. 1/10, 4/10, ...) cannot be exactly represented by finite binary floating point number (also by `float`).

Floating point representation – pros and cons

Generally, due to limited memory, no type can represent all numbers accurately.

Pros: A floating-point system can be used to represent numbers of wildly different orders of magnitude.
For example the same data type can store the distance between galaxies and the diameter of an atomic nucleus.

Cons: Many rational numbers cannot be exactly represented with finite memory. For instance:

- $1/3 = 0.333\ldots$ (and also e.g. 1/30, 10/3, ...) does not have an exact finite decimal floating point representation;
- $2/10 = 0.00110011\ldots_2$ (and also e.g. 1/10, 4/10, ...) cannot be exactly represented by finite binary floating point number (also by `float`).

Floating point representation – pros and cons

Generally, due to limited memory, no type can represent all numbers accurately.

Pros: A floating-point system can be used to represent numbers of wildly different orders of magnitude.
For example the same data type can store the distance between galaxies and the diameter of an atomic nucleus.

Cons: Many rational numbers cannot be exactly represented with finite memory. For instance:

- $1/3 = 0.333\ldots$ (and also e.g. 1/30, 10/3, ...) does not have an exact finite decimal floating point representation;
- $2/10 = 0.00110011\ldots_2$ (and also e.g. 1/10, 4/10, ...) cannot be exactly represented by finite binary floating point number (also by `float`).

Floating point representation – pros and cons

Generally, due to limited memory, no type can represent all numbers accurately.

Pros: A floating-point system can be used to represent numbers of wildly different orders of magnitude.
For example the same data type can store the distance between galaxies and the diameter of an atomic nucleus.

Cons: Many rational numbers cannot be exactly represented with finite memory. For instance:

- $1/3 = 0.333\ldots$ (and also e.g. 1/30, 10/3, ...) does not have an exact finite decimal floating point representation;
- $2/10 = 0.00110011\ldots_2$ (and also e.g. 1/10, 4/10, ...) cannot be exactly represented by finite binary floating point number (also by `float`).

Floating point representation – pros and cons

Generally, due to limited memory, no type can represent all numbers accurately.

Pros: A floating-point system can be used to represent numbers of wildly different orders of magnitude.
For example the same data type can store the distance between galaxies and the diameter of an atomic nucleus.

Cons: Many rational numbers cannot be exactly represented with finite memory. For instance:

- $1/3 = 0.333\ldots$ (and also e.g. 1/30, 10/3, ...) does not have an exact finite decimal floating point representation;
- $2/10 = 0.00110011\ldots_2$ (and also e.g. 1/10, 4/10, ...) cannot be exactly represented by finite binary floating point number (also by `float`).

Floating point representation – pros and cons

Generally, due to limited memory, no type can represent all numbers accurately.

Pros: A floating-point system can be used to represent numbers of wildly different orders of magnitude.
For example the same data type can store the distance between galaxies and the diameter of an atomic nucleus.

Cons: Many rational numbers cannot be exactly represented with finite memory. For instance:

- $1/3 = 0.333\ldots$ (and also e.g. $1/30$, $10/3$, ...) does not have an exact finite decimal floating point representation;
- $2/10 = 0.00110011\ldots_2$ (and also e.g. $1/10$, $4/10$, ...) cannot be exactly represented by finite binary floating point number (also by `float`).

Theorem: Let s, b, e, m be integers, $b \geq 2$ and $m \geq 1$. Number x of the form

$$s \cdot b^e$$

can also be represented in the form

$$z \cdot (bm)^f,$$

where z and f are integers.

Proof: If $e \geq 0$ than x is an integer and we can choose $z = x$ and $f = 0$. Than $z \cdot (bm)^f = x \cdot (bm)^0 = x$.
If $e < 0$ than m^{-e} is an integer and we can choose $z = sm^{-e}$ and $f = e$. Than $z \cdot (bm)^f = sm^{-e} \cdot (bm)^e = sm^{-e}b^e m^e = s \cdot b^e = x$.

Conclusion (for b=2 and m=5)

All finite binary floating-point numbers have exact and finite decimal floating-point representations.

Theorem: Let s, b, e, m be integers, $b \geq 2$ and $m \geq 1$. Number x of the form

$$s \cdot b^e$$

can also be represented in the form

$$z \cdot (bm)^f,$$

where z and f are integers.

Proof: If $e \geq 0$ than x is an integer and we can choose $z = x$ and $f = 0$. Than $z \cdot (bm)^f = x \cdot (bm)^0 = x$.
If $e < 0$ than m^{-e} is an integer and we can choose $z = sm^{-e}$ and $f = e$. Than $z \cdot (bm)^f = sm^{-e} \cdot (bm)^e = sm^{-e}b^e m^e = s \cdot b^e = x$.

Conclusion (for b=2 and m=5)

All finite binary floating-point numbers have exact and finite decimal floating-point representations.

Theorem: Let s, b, e, m be integers, $b \geq 2$ and $m \geq 1$. Number x of the form

$$s \cdot b^e$$

can also be represented in the form

$$z \cdot (bm)^f,$$

where z and f are integers.

Proof: If $e \geq 0$ than x is an integer and we can choose $z = x$ and $f = 0$. Than $z \cdot (bm)^f = x \cdot (bm)^0 = x$.
If $e < 0$ than m^{-e} is an integer and we can choose $z = sm^{-e}$ and $f = e$. Than $z \cdot (bm)^f = sm^{-e} \cdot (bm)^e = sm^{-e}b^e m^e = s \cdot b^e = x$.

Conclusion (for b=2 and m=5)

All finite binary floating-point numbers have exact and finite decimal floating-point representations.

Theorem: Let s, b, e, m be integers, $b \geq 2$ and $m \geq 1$. Number x of the form

$$s \cdot b^e$$

can also be represented in the form

$$z \cdot (bm)^f,$$

where z and f are integers.

Proof: If $e \geq 0$ than x is an integer and we can choose $z = x$ and $f = 0$. Than $z \cdot (bm)^f = x \cdot (bm)^0 = x$.
If $e < 0$ than m^{-e} is an integer and we can choose $z = sm^{-e}$ and $f = e$. Than $z \cdot (bm)^f = sm^{-e} \cdot (bm)^e = sm^{-e}b^e m^e = s \cdot b^e = x$.

Conclusion (for b=2 and m=5)

All finite binary floating-point numbers have exact and finite decimal floating-point representations.

Alternative representations of rational numbers

Python's standard library provides the following alternatives to `float`:

- the `decimal` module provides `Decimal` type and support for fast correctly-rounded decimal floating point arithmetic;
- the `fractions` module provides `Fraction` type and support for rational number arithmetic.

Both `Decimal` and `Fraction`:

- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, `pow`, `abs`, `round`;
- support conversions to/from built-in types like `float`, `int`, or `str` (`Decimal` can also be converted to `Fraction`);
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to `float`;
- in contrast to `float`, usually do not have special hardware support and work slower than `float`.

Alternative representations of rational numbers

Python's standard library provides the following alternatives to **`float`**:

- the `decimal` module provides `Decimal` type and support for fast correctly-rounded decimal floating point arithmetic;
- the `fractions` module provides `Fraction` type and support for rational number arithmetic.

Both `Decimal` and `Fraction`:

- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- support conversions to/from built-in types like **`float`**, **`int`**, or **`str`** (`Decimal` can also be converted to `Fraction`);
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**;
- in contrast to **`float`**, usually do not have special hardware support and work slower than **`float`**.

Alternative representations of rational numbers

Python's standard library provides the following alternatives to **`float`**:

- the `decimal` module provides `Decimal` type and support for fast correctly-rounded decimal floating point arithmetic;
- the `fractions` module provides `Fraction` type and support for rational number arithmetic.

Both `Decimal` and `Fraction`:

- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- support conversions to/from built-in types like **`float`**, **`int`**, or **`str`** (`Decimal` can also be converted to `Fraction`);
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**;
- in contrast to **`float`**, usually do not have special hardware support and work slower than **`float`**.

Alternative representations of rational numbers

Python's standard library provides the following alternatives to **`float`**:

- the `decimal` module provides `Decimal` type and support for fast correctly-rounded decimal floating point arithmetic;
- the `fractions` module provides `Fraction` type and support for rational number arithmetic.

Both `Decimal` and `Fraction`:

- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- support conversions to/from built-in types like **`float`**, **`int`**, or **`str`** (`Decimal` can also be converted to `Fraction`);
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**;
- in contrast to **`float`**, usually do not have special hardware support and work slower than **`float`**.

Alternative representations of rational numbers

Python's standard library provides the following alternatives to **`float`**:

- the `decimal` module provides `Decimal` type and support for fast correctly-rounded decimal floating point arithmetic;
- the `fractions` module provides `Fraction` type and support for rational number arithmetic.

Both `Decimal` and `Fraction`:

- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- support conversions to/from built-in types like **`float`**, **`int`**, or **`str`** (`Decimal` can also be converted to `Fraction`);
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**;
- in contrast to **`float`**, usually do not have special hardware support and work slower than **`float`**.

Alternative representations of rational numbers

Python's standard library provides the following alternatives to **`float`**:

- the `decimal` module provides `Decimal` type and support for fast correctly-rounded decimal floating point arithmetic;
- the `fractions` module provides `Fraction` type and support for rational number arithmetic.

Both `Decimal` and `Fraction`:

- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- support conversions to/from built-in types like **`float`**, **`int`**, or **`str`** (`Decimal` can also be converted to `Fraction`);
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**;
- in contrast to **`float`**, usually do not have special hardware support and work slower than **`float`**.

Alternative representations of rational numbers

Python's standard library provides the following alternatives to **`float`**:

- the `decimal` module provides `Decimal` type and support for fast correctly-rounded decimal floating point arithmetic;
- the `fractions` module provides `Fraction` type and support for rational number arithmetic.

Both `Decimal` and `Fraction`:

- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- support conversions to/from built-in types like **`float`**, **`int`**, or **`str`** (`Decimal` can also be converted to `Fraction`);
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**;
- in contrast to **`float`**, usually do not have special hardware support and work slower than **`float`**.

Alternative representations of rational numbers

Python's standard library provides the following alternatives to **`float`**:

- the `decimal` module provides `Decimal` type and support for fast correctly-rounded decimal floating point arithmetic;
- the `fractions` module provides `Fraction` type and support for rational number arithmetic.

Both `Decimal` and `Fraction`:

- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- support conversions to/from built-in types like **`float`**, **`int`**, or **`str`** (`Decimal` can also be converted to `Fraction`);
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**;
- in contrast to **`float`**, usually do not have special hardware support and work slower than **`float`**.

Decimal

The official Python documentation enumerates that compared to the `float`, the `Decimal` is especially helpful for:

- **financial applications** and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

Decimal

The official Python documentation enumerates that compared to the `float`, the `Decimal` is especially helpful for:

- **financial applications** and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

Decimal

The official Python documentation enumerates that compared to the `float`, the `Decimal` is especially helpful for:

- **financial applications** and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

Decimal

The official Python documentation enumerates that compared to the `float`, the `Decimal` is especially helpful for:

- **financial applications** and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

Decimal

The official Python documentation enumerates that compared to the `float`, the `Decimal` is especially helpful for:

- **financial applications** and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

Decimal

The official Python documentation enumerates that compared to the `float`, the `Decimal` is especially helpful for:

- **financial applications** and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

Decimal – importing and constructing – examples

In order to use the `Decimal` type, import it first:

```
from decimal import Decimal as D
```

(due to `as D` fragment, we can later use the shorter name `D` instead of the full name `Decimal`).

Next, execute the following expressions:

`D(10)`	constructs `Decimal` from `int`;
`D(0.2)`	constructs `Decimal` from `float`, but `float` cannot represents the number 0.2 exactly;
`D('0.2')`	constructing from `string` works as expected;
`D('0.20')`	differs from the previous one since trailing zeros are kept (and tracked) to indicate significance;
`D('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`D('2.0e-1')`	scientific notation with one more trailing zero.

Exercise: calculate and notice a difference in trailing zeros of: `D('1.30') * D('1.20')` and `D('1.3') * D('1.2')`

Decimal – importing and constructing – examples

In order to use the `Decimal` type, import it first:

```
from decimal import Decimal as D
```

(due to `as D` fragment, we can later use the shorter name `D` instead of the full name `Decimal`).

Next, execute the following expressions:

`D(10)`	constructs `Decimal` from **`int`**;
`D(0.2)`	constructs `Decimal` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`D('0.2')`	constructing from **`str`**ing works as expected;
`D('0.20')`	differs from the previous one since trailing zeros are kept (and tracked) to indicate significance;
`D('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`D('2.0e-1')`	scientific notation with one more trailing zero.

Exercise: calculate and notice a difference in trailing zeros of: `D('1.30') * D('1.20')` and `D('1.3') * D('1.2')`

Decimal – importing and constructing – examples

In order to use the `Decimal` type, import it first:

```
from decimal import Decimal as D
```

(due to `as D` fragment, we can later use the shorter name `D` instead of the full name `Decimal`).

Next, execute the following expressions:

`D(10)`	constructs `Decimal` from `int`;
`D(0.2)`	constructs `Decimal` from `float`, but `float` cannot represents the number 0.2 exactly;
`D('0.2')`	constructing from `str`ing works as expected;
`D('0.20')`	differs from the previous one since trailing zeros are kept (and tracked) to indicate significance;
`D('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`D('2.0e-1')`	scientific notation with one more trailing zero.

Exercise: calculate and notice a difference in trailing zeros of: `D('1.30') * D('1.20')` and `D('1.3') * D('1.2')`

Decimal – importing and constructing – examples

In order to use the `Decimal` type, import it first:

```
from decimal import Decimal as D
```

(due to `as D` fragment, we can later use the shorter name `D` instead of the full name `Decimal`).

Next, execute the following expressions:

`D(10)`	constructs `Decimal` from **`int`**;
`D(0.2)`	constructs `Decimal` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`D('0.2')`	constructing from **`str`**ing works as expected;
`D('0.20')`	differs from the previous one since trailing zeros are kept (and tracked) to indicate significance;
`D('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`D('2.0e-1')`	scientific notation with one more trailing zero.

Exercise: calculate and notice a difference in trailing zeros of: `D('1.30') * D('1.20')` and `D('1.3') * D('1.2')`

Decimal – importing and constructing – examples

In order to use the `Decimal` type, import it first:

```
from decimal import Decimal as D
```

(due to `as D` fragment, we can later use the shorter name `D` instead of the full name `Decimal`).

Next, execute the following expressions:

`D(10)`	constructs `Decimal` from **`int`**;
`D(0.2)`	constructs `Decimal` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`D('0.2')`	constructing from **`str`**ing works as expected;
`D('0.20')`	differs from the previous one since trailing zeros are kept (and tracked) to indicate significance;
`D('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`D('2.0e-1')`	scientific notation with one more trailing zero.

Exercise: calculate and notice a difference in trailing zeros of: `D('1.30') * D('1.20')` and `D('1.3') * D('1.2')`

Decimal – importing and constructing – examples

In order to use the `Decimal` type, import it first:

```
from decimal import Decimal as D
```

(due to `as D` fragment, we can later use the shorter name `D` instead of the full name `Decimal`).

Next, execute the following expressions:

`D(10)`	constructs `Decimal` from **`int`**;
`D(0.2)`	constructs `Decimal` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`D('0.2')`	constructing from **`str`**ing works as expected;
`D('0.20')`	differs from the previous one since trailing zeros are kept (and tracked) to indicate significance;
`D('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`D('2.0e-1')`	scientific notation with one more trailing zero.

Exercise: calculate and notice a difference in trailing zeros of: `D('1.30') * D('1.20')` and `D('1.3') * D('1.2')`

Decimal – importing and constructing – examples

In order to use the `Decimal` type, import it first:

```
from decimal import Decimal as D
```

(due to `as D` fragment, we can later use the shorter name `D` instead of the full name `Decimal`).

Next, execute the following expressions:

`D(10)`	constructs `Decimal` from **`int`**;
`D(0.2)`	constructs `Decimal` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`D('0.2')`	constructing from **`str`**ing works as expected;
`D('0.20')`	differs from the previous one since trailing zeros are kept (and tracked) to indicate significance;
`D('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`D('2.0e-1')`	scientific notation with one more trailing zero.

Exercise: calculate and notice a difference in trailing zeros of: `D('1.30') * D('1.20')` and `D('1.3') * D('1.2')`

Decimal – importing and constructing – examples

In order to use the `Decimal` type, import it first:

```
from decimal import Decimal as D
```

(due to `as D` fragment, we can later use the shorter name `D` instead of the full name `Decimal`).

Next, execute the following expressions:

`D(10)`	constructs `Decimal` from **`int`**;
`D(0.2)`	constructs `Decimal` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`D('0.2')`	constructing from **`str`**ing works as expected;
`D('0.20')`	differs from the previous one since trailing zeros are kept (and tracked) to indicate significance;
`D('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`D('2.0e-1')`	scientific notation with one more trailing zero.

Exercise: calculate and notice a difference in trailing zeros of:
`D('1.30') * D('1.20')` and `D('1.3') * D('1.2')`

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a `float`;
`d+5`	is `Decimal`, a common type for `Decimal` and `int`;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float(d)+f`	enforces using `float`;
`d+D(f)`	enforces using `Decimal`;
`abs(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: `help(D)`):

1. $\log_{10}(0.1)$
2. $\sqrt{2}$

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a `float`;
`d+5`	is `Decimal`, a common type for `Decimal` and `int`;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float(d)+f`	enforces using `float`;
`d+D(f)`	enforces using `Decimal`;
`abs(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: `help(D)`):

1. $\log_{10}(0.1)$
2. $\sqrt{2}$

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a `float`;
`d+5`	is `Decimal`, a common type for `Decimal` and `int`;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float(d)+f`	enforces using `float`;
`d+D(f)`	enforces using `Decimal`;
`abs(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: `help(D)`):

1. $\log_{10}(0.1)$
2. $\sqrt{2}$

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a `float`;
`d+5`	is `Decimal`, a common type for `Decimal` and `int`;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float(d)+f`	enforces using `float`;
`d+D(f)`	enforces using `Decimal`;
`abs(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: `help(D)`):

1. $\log_{10}(0.1)$
2. $\sqrt{2}$

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a `float`;
`d+5`	is `Decimal`, a common type for `Decimal` and `int`;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float(d)+f`	enforces using `float`;
`d+D(f)`	enforces using `Decimal`;
`abs(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: `help(D)`):

1. $\log_{10}(0.1)$
2. $\sqrt{2}$

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a `float`;
`d+5`	is `Decimal`, a common type for `Decimal` and `int`;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float(d)+f`	enforces using `float`;
`d+D(f)`	enforces using `Decimal`;
`abs(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: `help(D)`):

1. $\log_{10}(0.1)$
2. $\sqrt{2}$

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a `float`;
`d+5`	is `Decimal`, a common type for `Decimal` and `int`;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float(d)+f`	enforces using `float`;
`d+D(f)`	enforces using `Decimal`;
`abs(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: `help(D)`):

1. $\log_{10}(0.1)$
2. $\sqrt{2}$

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a `float`;
`d+5`	is `Decimal`, a common type for `Decimal` and `int`;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float(d)+f`	enforces using `float`;
`d+D(f)`	enforces using `Decimal`;
`abs(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: `help(D)`):

1. $\log_{10}(0.1)$
2. $\sqrt{2}$

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a `float`;
`d+5`	is `Decimal`, a common type for `Decimal` and `int`;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float(d)+f`	enforces using `float`;
`d+D(f)`	enforces using `Decimal`;
`abs(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: `help(D)`):

1. $\log_{10}(0.1)$
2. $\sqrt{2}$

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a `float`;
`d+5`	is `Decimal`, a common type for `Decimal` and `int`;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float(d)+f`	enforces using `float`;
`d+D(f)`	enforces using `Decimal`;
`abs(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: `help(D)`):

1. $\log_{10}(0.1)$
2. $\sqrt{2}$

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a **`float`**;
`d+5`	is `Decimal`, a common type for `Decimal` and **`int`**;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float``(d)+f`	enforces using **`float`**;
`d+D(f)`	enforces using `Decimal`;
`abs``(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use **`float`** for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: **`help`**`(D)`):

1. $\log_{10}(0.1)$
2. $\sqrt{2}$

Decimal – examples and exercises

Execute:

```
import math as m
```

`d=D(-1)`	let `d` refer to a `Decimal` instance;
`f=1.5`	and `f` to a **`float`**;
`d+5`	is `Decimal`, a common type for `Decimal` and **`int`**;
`d+f`	raises `TypeError`, as it is unclear which type should be used for calculations;
`float``(d)+f`	enforces using **`float`**;
`d+D(f)`	enforces using `Decimal`;
`abs``(d)`	gives `Decimal`; built-in functions just work;
`m.exp(d)`	most functions from the `math` module convert their arguments and use **`float`** for calculations;
`d.sqrt()`	the `Decimal` methods avoid such conversions.

Calculate using `Decimal` (hint: **`help`**`(D)`):

1. $\log_{10}(0.1)$ `D('0.1').log10()` is more accurate than `D(0.1).log10()` or `m.log10(D('0.1'))`
2. $\sqrt{2}$ `D(2).sqrt()` or `D(2)**D('0.5')`

Decimal – a real life example

Calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point.

The difference becomes significant if the results are rounded to the nearest cent:

`round(D('0.70') * D('1.05'), 2)` gives 0.74,

`round(0.70 * 1.05, 2)` gives 0.73.

(the example comes from the official Python documentation)

Exercise: calculate remainder after division of 1 by 0.1.

Compare the accuracy of decimal (`Decimal`) and binary (`float`) floating point calculations in this task.

Decimal – a real life example

Calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point.
The difference becomes significant if the results are rounded to the nearest cent:
`round(D('0.70') * D('1.05'), 2)` gives 0.74,
`round(0.70 * 1.05, 2)` gives 0.73.
(the example comes from the official Python documentation)

Exercise: calculate remainder after division of 1 by 0.1.
Compare the accuracy of decimal (`Decimal`) and binary (`float`) floating point calculations in this task.

Decimal – a real life example

Calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point.
The difference becomes significant if the results are rounded to the nearest cent:
`round(D('0.70') * D('1.05'), 2)` gives 0.74,
`round(0.70 * 1.05, 2)` gives 0.73.
(the example comes from the official Python documentation)

Exercise: calculate remainder after division of 1 by 0.1.
Compare the accuracy of decimal (`Decimal`) and binary (`float`) floating point calculations in this task.

Decimal – a real life example

Calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point.
The difference becomes significant if the results are rounded to the nearest cent:
`round(D('0.70') * D('1.05'), 2)` gives 0.74,
`round(0.70 * 1.05, 2)` gives 0.73.
(the example comes from the official Python documentation)

Exercise: calculate remainder after division of 1 by 0.1.
Compare the accuracy of decimal (`Decimal`) and binary (`float`) floating point calculations in this task.
`D(1) % D('0.1')` gives exact result 0.
`1 % 0.1` gives very inaccurate result.

Decimal – context

The `decimal` module allows for adjusting the context which is an environment for arithmetic operations and specifies their precision, rounding rules, and other properties.

Context precision and rounding only come into play during arithmetic operations (not while constructing a new `Decimal`).

Execute:

```
from decimal import getcontext
```

`getcontext()`	displays the current context;
`getcontext().prec=100`	sets a new precision – the maximum number of significant decimal digits;
`D(3).sqrt()`	displays 1 integer digit and 99 fractional digits of $\sqrt{3}$.

Exercise: calculate 200 significant decimal figures of e^{500}.

Decimal – context

The `decimal` module allows for adjusting the context which is an environment for arithmetic operations and specifies their precision, rounding rules, and other properties.
Context precision and rounding only come into play during arithmetic operations (not while constructing a new `Decimal`).

Execute:

```
from decimal import getcontext
```

`getcontext()`	displays the current context;
`getcontext().prec=100`	sets a new precision – the maximum number of significant decimal digits;
`D(3).sqrt()`	displays 1 integer digit and 99 fractional digits of $\sqrt{3}$.

Exercise: calculate 200 significant decimal figures of e^{500}.

Decimal – context

The `decimal` module allows for adjusting the context which is an environment for arithmetic operations and specifies their precision, rounding rules, and other properties.
Context precision and rounding only come into play during arithmetic operations (not while constructing a new `Decimal`).

Execute:

```
from decimal import getcontext
```

`getcontext()`	displays the current context;
`getcontext().prec=100`	sets a new precision – the maximum number of significant decimal digits;
`D(3).sqrt()`	displays 1 integer digit and 99 fractional digits of $\sqrt{3}$.

Exercise: calculate 200 significant decimal figures of e^{500}.

Decimal – context

The `decimal` module allows for adjusting the context which is an environment for arithmetic operations and specifies their precision, rounding rules, and other properties.
Context precision and rounding only come into play during arithmetic operations (not while constructing a new `Decimal`).

Execute:

`from decimal import getcontext`	
`getcontext()`	displays the current context;
`getcontext().prec=100`	sets a new precision – the maximum number of significant decimal digits;
`D(3).sqrt()`	displays 1 integer digit and 99 fractional digits of $\sqrt{3}$.

Exercise: calculate 200 significant decimal figures of e^{500}.

Decimal – context

The `decimal` module allows for adjusting the context which is an environment for arithmetic operations and specifies their precision, rounding rules, and other properties.
Context precision and rounding only come into play during arithmetic operations (not while constructing a new `Decimal`).

Execute:

`from decimal import getcontext`	
`getcontext()`	displays the current context;
`getcontext().prec=100`	sets a new precision – the maximum number of significant decimal digits;
`D(3).sqrt()`	displays 1 integer digit and 99 fractional digits of $\sqrt{3}$.

Exercise: calculate 200 significant decimal figures of e^{500}.

Decimal – context

The `decimal` module allows for adjusting the context which is an environment for arithmetic operations and specifies their precision, rounding rules, and other properties.
Context precision and rounding only come into play during arithmetic operations (not while constructing a new `Decimal`).

Execute:

```
from decimal import getcontext
```

`getcontext()`	displays the current context;
`getcontext().prec=100`	sets a new precision – the maximum number of significant decimal digits;
`D(3).sqrt()`	displays 1 integer digit and 99 fractional digits of $\sqrt{3}$.

Exercise: calculate 200 significant decimal figures of e^{500}.

Decimal – context

The `decimal` module allows for adjusting the context which is an environment for arithmetic operations and specifies their precision, rounding rules, and other properties.
Context precision and rounding only come into play during arithmetic operations (not while constructing a new `Decimal`).

Execute:

`from decimal import getcontext`	
`getcontext()`	displays the current context;
`getcontext().prec=100`	sets a new precision – the maximum number of significant decimal digits;
`D(3).sqrt()`	displays 1 integer digit and 99 fractional digits of $\sqrt{3}$.

Exercise: calculate 200 significant decimal figures of e^{500}.

Decimal – context

The `decimal` module allows for adjusting the context which is an environment for arithmetic operations and specifies their precision, rounding rules, and other properties.
Context precision and rounding only come into play during arithmetic operations (not while constructing a new `Decimal`).

Execute:

```
from decimal import getcontext
```

`getcontext()`	displays the current context;
`getcontext().prec=100`	sets a new precision – the maximum number of significant decimal digits;
`D(3).sqrt()`	displays 1 integer digit and 99 fractional digits of $\sqrt{3}$.

Exercise: calculate 200 significant decimal figures of e^{500}.
`getcontext().prec=200`
`D(500).exp()` (not `D(math.e) ** 500` since `math.e` is **`float`** and not accurate enough)

Fraction

The `Fraction` (from the `fractions` module):

- represents a rational number (implements `numbers.Rational` which is an abstract base class) as a fraction $\frac{\text{numerator}}{\text{denominator}}$ of **`int`**egers;
- consists of two (arbitrary-precision) **`int`**egers, accessible via its members: `numerator` and `denominator`;
- has precision limited only by the available memory;
- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**.

Exercise: import the `Fraction` type from the `fractions` module and make this type accessible via the `F` name.

Fraction

The `Fraction` (from the `fractions` module):

- represents a rational number (implements `numbers.Rational` which is an abstract base class) as a fraction $\frac{\text{numerator}}{\text{denominator}}$ of **`int`**egers;
- consists of two (arbitrary-precision) **`int`**egers, accessible via its members: `numerator` and `denominator`;
- has precision limited only by the available memory;
- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**.

Exercise: import the `Fraction` type from the `fractions` module and make this type accessible via the `F` name.

Fraction

The `Fraction` (from the `fractions` module):

- represents a rational number (implements `numbers.Rational` which is an abstract base class) as a fraction $\frac{\text{numerator}}{\text{denominator}}$ of **`int`**egers;
- consists of two (arbitrary-precision) **`int`**egers, accessible via its members: `numerator` and `denominator`;
- has precision limited only by the available memory;
- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**.

Exercise: import the `Fraction` type from the `fractions` module and make this type accessible via the `F` name.

Fraction

The `Fraction` (from the `fractions` module):

- represents a rational number (implements `numbers.Rational` which is an abstract base class) as a fraction $\frac{\text{numerator}}{\text{denominator}}$ of **`int`**egers;
- consists of two (arbitrary-precision) **`int`**egers, accessible via its members: `numerator` and `denominator`;
- has precision limited only by the available memory;
- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**.

Exercise: import the `Fraction` type from the `fractions` module and make this type accessible via the `F` name.

Fraction

The `Fraction` (from the `fractions` module):

- represents a rational number (implements `numbers.Rational` which is an abstract base class) as a fraction $\frac{\text{numerator}}{\text{denominator}}$ of **`int`**egers;
- consists of two (arbitrary-precision) **`int`**egers, accessible via its members: `numerator` and `denominator`;
- has precision limited only by the available memory;
- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**.

Exercise: import the `Fraction` type from the `fractions` module and make this type accessible via the `F` name.

Fraction

The `Fraction` (from the `fractions` module):

- represents a rational number (implements `numbers.Rational` which is an abstract base class) as a fraction $\frac{\text{numerator}}{\text{denominator}}$ of **`int`**egers;
- consists of two (arbitrary-precision) **`int`**egers, accessible via its members: `numerator` and `denominator`;
- has precision limited only by the available memory;
- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**.

Exercise: import the `Fraction` type from the `fractions` module and make this type accessible via the `F` name.

Fraction

The `Fraction` (from the `fractions` module):

- represents a rational number (implements `numbers.Rational` which is an abstract base class) as a fraction $\frac{\text{numerator}}{\text{denominator}}$ of **`int`**egers;
- consists of two (arbitrary-precision) **`int`**egers, accessible via its members: `numerator` and `denominator`;
- has precision limited only by the available memory;
- support all standard operations, like: `+`, `-`, `*`, `/`, `//`, `**`, **`pow`**, **`abs`**, **`round`**;
- can be passed to functions provided by the `math` module, but in most cases it involves automatic conversion to **`float`**.

Exercise: import the `Fraction` type from the `fractions` module and make this type accessible via the `F` name.

Execute:

```
from fractions import Fraction as F
```

Fraction – constructing – examples

Execute:

`F(-7)`	constructs `Fraction` $-7/1$ from **`int`**;
`F()`	represents 0 as $0/1$; same as `F(0)`;
`F(1, 5)`	constructs `Fraction` $1/5$ from two **`int`**egers: numerator and denominator;
`F(2, 10)`	also $1/5$; the simplest (reduced) form is used;
`F(0.2)`	constructs `Fraction` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`F('0.2')`	constructing from **`str`**ing works as expected;
`F('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`F('2/10')`	such **`str`**ings are also accepted;
`F(D('0.2'))`	constructs `Fraction` from `Decimal` (but opposite conversion is not supported); note that this expression relies on earlier **`from`** `decimal` **`import`** `Decimal` **`as`** `D`

Fraction – constructing – examples

Execute:

`F(-7)`	constructs `Fraction` $-7/1$ from **`int`**;
`F()`	represents 0 as $0/1$; same as `F(0)`;
`F(1, 5)`	constructs `Fraction` $1/5$ from two **`int`**egers: numerator and denominator;
`F(2, 10)`	also $1/5$; the simplest (reduced) form is used;
`F(0.2)`	constructs `Fraction` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`F('0.2')`	constructing from **`str`**ing works as expected;
`F('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`F('2/10')`	such **`str`**ings are also accepted;
`F(D('0.2'))`	constructs `Fraction` from `Decimal` (but opposite conversion is not supported); note that this expression relies on earlier **`from`** `decimal` **`import`** `Decimal` **`as`** `D`

Fraction – constructing – examples

Execute:

`F(-7)`	constructs `Fraction` $-7/1$ from **`int`**;
`F()`	represents 0 as $0/1$; same as `F(0)`;
`F(1, 5)`	constructs `Fraction` $1/5$ from two **`int`**egers: numerator and denominator;
`F(2, 10)`	also $1/5$; the simplest (reduced) form is used;
`F(0.2)`	constructs `Fraction` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`F('0.2')`	constructing from **`str`**ing works as expected;
`F('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`F('2/10')`	such **`str`**ings are also accepted;
`F(D('0.2'))`	constructs `Fraction` from `Decimal` (but opposite conversion is not supported); note that this expression relies on earlier **`from`** `decimal` **`import`** `Decimal` **`as`** `D`

Fraction – constructing – examples

Execute:

`F(-7)`	constructs `Fraction` $-7/1$ from **`int`**;
`F()`	represents 0 as $0/1$; same as `F(0)`;
`F(1, 5)`	constructs `Fraction` $1/5$ from two **`int`**egers: numerator and denominator;
`F(2, 10)`	also $1/5$; the simplest (reduced) form is used;
`F(0.2)`	constructs `Fraction` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`F('0.2')`	constructing from **`str`**ing works as expected;
`F('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`F('2/10')`	such **`str`**ings are also accepted;
`F(D('0.2'))`	constructs `Fraction` from `Decimal` (but opposite conversion is not supported); note that this expression relies on earlier **`from`** `decimal` **`import`** `Decimal` **`as`** `D`

Fraction – constructing – examples

Execute:

`F(-7)`	constructs `Fraction` $-7/1$ from **`int`**;
`F()`	represents 0 as $0/1$; same as `F(0)`;
`F(1, 5)`	constructs `Fraction` $1/5$ from two **`int`**egers: numerator and denominator;
`F(2, 10)`	also $1/5$; the simplest (reduced) form is used;
`F(0.2)`	constructs `Fraction` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`F('0.2')`	constructing from **`str`**ing works as expected;
`F('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`F('2/10')`	such **`str`**ings are also accepted;
`F(D('0.2'))`	constructs `Fraction` from `Decimal` (but opposite conversion is not supported); note that this expression relies on earlier **`from`** `decimal` **`import`** `Decimal` **`as`** `D`

Fraction – constructing – examples

Execute:

`F(-7)`	constructs `Fraction` $-7/1$ from **`int`**;
`F()`	represents 0 as $0/1$; same as `F(0)`;
`F(1, 5)`	constructs `Fraction` $1/5$ from two **`int`**egers: numerator and denominator;
`F(2, 10)`	also $1/5$; the simplest (reduced) form is used;
`F(0.2)`	constructs `Fraction` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`F('0.2')`	constructing from **`str`**ing works as expected;
`F('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`F('2/10')`	such **`str`**ings are also accepted;
`F(D('0.2'))`	constructs `Fraction` from `Decimal` (but opposite conversion is not supported); note that this expression relies on earlier **`from`** `decimal` **`import`** `Decimal` **`as`** `D`

Fraction – constructing – examples

Execute:

`F(-7)`	constructs `Fraction` $-7/1$ from **`int`**;
`F()`	represents 0 as $0/1$; same as `F(0)`;
`F(1, 5)`	constructs `Fraction` $1/5$ from two **`int`**egers: numerator and denominator;
`F(2, 10)`	also $1/5$; the simplest (reduced) form is used;
`F(0.2)`	constructs `Fraction` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`F('0.2')`	constructing from **`str`**ing works as expected;
`F('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`F('2/10')`	such **`str`**ings are also accepted;
`F(D('0.2'))`	constructs `Fraction` from `Decimal` (but opposite conversion is not supported); note that this expression relies on earlier **`from`** `decimal` **`import`** `Decimal` **`as`** `D`

Fraction – constructing – examples

Execute:

`F(-7)`	constructs `Fraction` $-7/1$ from **`int`**;
`F()`	represents 0 as $0/1$; same as `F(0)`;
`F(1, 5)`	constructs `Fraction` $1/5$ from two **`int`**egers: numerator and denominator;
`F(2, 10)`	also $1/5$; the simplest (reduced) form is used;
`F(0.2)`	constructs `Fraction` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`F('0.2')`	constructing from **`str`**ing works as expected;
`F('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`F('2/10')`	such **`str`**ings are also accepted;
`F(D('0.2'))`	constructs `Fraction` from `Decimal` (but opposite conversion is not supported); note that this expression relies on earlier **`from`** `decimal` **`import`** `Decimal` **`as`** `D`

Fraction – constructing – examples

Execute:

`F(-7)`	constructs `Fraction` $-7/1$ from **`int`**;
`F()`	represents 0 as $0/1$; same as `F(0)`;
`F(1, 5)`	constructs `Fraction` $1/5$ from two **`int`**egers: numerator and denominator;
`F(2, 10)`	also $1/5$; the simplest (reduced) form is used;
`F(0.2)`	constructs `Fraction` from **`float`**, but **`float`** cannot represents the number 0.2 exactly;
`F('0.2')`	constructing from **`str`**ing works as expected;
`F('2e-1')`	uses scientific notation, *2e-1* means $2 \cdot 10^{-1}$;
`F('2/10')`	such **`str`**ings are also accepted;
`F(D('0.2'))`	constructs `Fraction` from `Decimal` (but opposite conversion is not supported); note that this expression relies on earlier **`from`** `decimal` **`import`** `Decimal` **`as`** `D`

Fraction – calculating – examples

(We suppose `import math as m` and
`from decimal import Decimal as D`.)

Execute:

`f=-F(1,2)`	let `f` refer to a `Fraction` instance;
`d=D('1.2')`	and `d` to a `Decimal` instance;
`f+3`	`Fraction` plus `int` gives `Fraction`;
`f+1.1`	but `Fraction` plus `float` gives `float`;
`f+d`	raises `TypeError`;
`D(f)+d`	raises `TypeError`: conversion from `Fraction` to `Decimal` is not supported;
`f+F(d)`	but the opposite conversion is supported;
`abs(f)`	gives `Fraction`; built-in functions just work;
`m.exp(f)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`str(f)`	conversion to `str`ing (`'-1/2'`).

Fraction – calculating – examples

(We suppose `import math as m` and
`from decimal import Decimal as D`.)

Execute:

`f=-F(1,2)`	let `f` refer to a `Fraction` instance;
`d=D('1.2')`	and `d` to a `Decimal` instance;
`f+3`	`Fraction` plus `int` gives `Fraction`;
`f+1.1`	but `Fraction` plus `float` gives `float`;
`f+d`	raises `TypeError`;
`D(f)+d`	raises `TypeError`: conversion from `Fraction` to `Decimal` is not supported;
`f+F(d)`	but the opposite conversion is supported;
`abs(f)`	gives `Fraction`; built-in functions just work;
`m.exp(f)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`str(f)`	conversion to `str`ing (`'-1/2'`).

Fraction – calculating – examples

(We suppose `import math as m` and
`from decimal import Decimal as D`.)

Execute:

`f=-F(1,2)`	let `f` refer to a `Fraction` instance;
`d=D('1.2')`	and `d` to a `Decimal` instance;
`f+3`	`Fraction` plus `int` gives `Fraction`;
`f+1.1`	but `Fraction` plus `float` gives `float`;
`f+d`	raises `TypeError`;
`D(f)+d`	raises `TypeError`: conversion from `Fraction` to `Decimal` is not supported;
`f+F(d)`	but the opposite conversion is supported;
`abs(f)`	gives `Fraction`; built-in functions just work;
`m.exp(f)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`str(f)`	conversion to `str`ing (`'-1/2'`).

Fraction – calculating – examples

(We suppose `import math as m` and
`from decimal import Decimal as D`.)

Execute:

`f=-F(1,2)`	let `f` refer to a `Fraction` instance;
`d=D('1.2')`	and `d` to a `Decimal` instance;
`f+3`	`Fraction` plus `int` gives `Fraction`;
`f+1.1`	but `Fraction` plus `float` gives `float`;
`f+d`	raises `TypeError`;
`D(f)+d`	raises `TypeError`: conversion from `Fraction` to `Decimal` is not supported;
`f+F(d)`	but the opposite conversion is supported;
`abs(f)`	gives `Fraction`; built-in functions just work;
`m.exp(f)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`str(f)`	conversion to `str`ing (`'-1/2'`).

Fraction – calculating – examples

(We suppose `import math as m` and
`from decimal import Decimal as D`.)

Execute:

`f=-F(1,2)`	let `f` refer to a `Fraction` instance;
`d=D('1.2')`	and `d` to a `Decimal` instance;
`f+3`	`Fraction` plus **`int`** gives `Fraction`;
`f+1.1`	but `Fraction` plus **`float`** gives **`float`**;
`f+d`	raises `TypeError`;
`D(f)+d`	raises `TypeError`: conversion from `Fraction` to `Decimal` is not supported;
`f+F(d)`	but the opposite conversion is supported;
`abs``(f)`	gives `Fraction`; built-in functions just work;
`m.exp(f)`	most functions from the `math` module convert their arguments and use **`float`** for calculations;
`str``(f)`	conversion to **`str`**ing (`'-1/2'`).

Fraction – calculating – examples

(We suppose `import math as m` and
`from decimal import Decimal as D`.)

Execute:

`f=-F(1,2)`	let `f` refer to a `Fraction` instance;
`d=D('1.2')`	and `d` to a `Decimal` instance;
`f+3`	`Fraction` plus `int` gives `Fraction`;
`f+1.1`	but `Fraction` plus `float` gives `float`;
`f+d`	raises `TypeError`;
`D(f)+d`	raises `TypeError`: conversion from `Fraction` to `Decimal` is not supported;
`f+F(d)`	but the opposite conversion is supported;
`abs(f)`	gives `Fraction`; built-in functions just work;
`m.exp(f)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`str(f)`	conversion to `str`ing (`'-1/2'`).

Fraction – calculating – examples

(We suppose `import math as m` and
`from decimal import Decimal as D`.)

Execute:

`f=-F(1,2)`	let `f` refer to a `Fraction` instance;
`d=D('1.2')`	and `d` to a `Decimal` instance;
`f+3`	`Fraction` plus `int` gives `Fraction`;
`f+1.1`	but `Fraction` plus `float` gives `float`;
`f+d`	raises `TypeError`;
`D(f)+d`	raises `TypeError`: conversion from `Fraction` to `Decimal` is not supported;
`f+F(d)`	but the opposite conversion is supported;
`abs(f)`	gives `Fraction`; built-in functions just work;
`m.exp(f)`	most functions from the `math` module convert their arguments and use `float` for calculations;
`str(f)`	conversion to `str`ing (`'-1/2'`).

Fraction – calculating – examples

(We suppose `import math as m` and
`from decimal import Decimal as D`.)

Execute:

`f=-F(1,2)`	let `f` refer to a `Fraction` instance;
`d=D('1.2')`	and `d` to a `Decimal` instance;
`f+3`	`Fraction` plus **`int`** gives `Fraction`;
`f+1.1`	but `Fraction` plus **`float`** gives **`float`**;
`f+d`	raises `TypeError`;
`D(f)+d`	raises `TypeError`: conversion from `Fraction` to `Decimal` is not supported;
`f+F(d)`	but the opposite conversion is supported;
`abs``(f)`	gives `Fraction`; built-in functions just work;
`m.exp(f)`	most functions from the `math` module convert their arguments and use **`float`** for calculations;
`str``(f)`	conversion to **`str`**ing (`'-1/2'`).

Fraction – calculating – examples

(We suppose `import math as m` and
`from decimal import Decimal as D`.)

Execute:

`f=-F(1,2)`	let `f` refer to a `Fraction` instance;
`d=D('1.2')`	and `d` to a `Decimal` instance;
`f+3`	`Fraction` plus **`int`** gives `Fraction`;
`f+1.1`	but `Fraction` plus **`float`** gives **`float`**;
`f+d`	raises `TypeError`;
`D(f)+d`	raises `TypeError`: conversion from `Fraction` to `Decimal` is not supported;
`f+F(d)`	but the opposite conversion is supported;
`abs``(f)`	gives `Fraction`; built-in functions just work;
`m.exp(f)`	most functions from the `math` module convert their arguments and use **`float`** for calculations;
`str``(f)`	conversion to **`str`**ing (`'-1/2'`).

Fraction – calculating – examples

(We suppose `import math as m` and
`from decimal import Decimal as D`.)

Execute:

`f=-F(1,2)`	let `f` refer to a `Fraction` instance;
`d=D('1.2')`	and `d` to a `Decimal` instance;
`f+3`	`Fraction` plus **`int`** gives `Fraction`;
`f+1.1`	but `Fraction` plus **`float`** gives **`float`**;
`f+d`	raises `TypeError`;
`D(f)+d`	raises `TypeError`: conversion from `Fraction` to `Decimal` is not supported;
`f+F(d)`	but the opposite conversion is supported;
`abs``(f)`	gives `Fraction`; built-in functions just work;
`m.exp(f)`	most functions from the `math` module convert their arguments and use **`float`** for calculations;
`str``(f)`	conversion to **`str`**ing (`'-1/2'`).

Fraction – calculating – examples

(We suppose `import math as m` and
`from decimal import Decimal as D`.)

Execute:

`f=-F(1,2)`	let `f` refer to a `Fraction` instance;
`d=D('1.2')`	and `d` to a `Decimal` instance;
`f+3`	`Fraction` plus **`int`** gives `Fraction`;
`f+1.1`	but `Fraction` plus **`float`** gives **`float`**;
`f+d`	raises `TypeError`;
`D(f)+d`	raises `TypeError`: conversion from `Fraction` to `Decimal` is not supported;
`f+F(d)`	but the opposite conversion is supported;
`abs``(f)`	gives `Fraction`; built-in functions just work;
`m.exp(f)`	most functions from the `math` module convert their arguments and use **`float`** for calculations;
`str``(f)`	conversion to **`str`**ing (`'-1/2'`).

Fraction – exercises

Calculate using `Fraction`:

1. $1/2 + 2/3$
2. $14\frac{4}{7} \cdot 11\frac{2}{19}$
3. 0.8^3

Approximate the following numbers by the fractions whose denominators are at most 81 (hint: `Fraction` has the `limit_denominator` method):

1. $55/703$
2. $3\frac{71}{501}$
3. π
4. 1.234

Fraction – exercises

Calculate using `Fraction`:

1. $1/2 + 2/3$ `F(1,2) + F(2,3)`
2. $14\frac{4}{7} \cdot 11\frac{2}{19}$ `(14+F(4,7)) * (11+F(2,19))`
3. 0.8^3 `F('0.8') ** 3` or `F(8,10) ** 3`
   are better than `F(0.8) ** 3`

Approximate the following numbers by the fractions whose denominators are at most 81 (hint: `Fraction` has the `limit_denominator` method):

1. $55/703$
2. $3\frac{71}{501}$
3. π
4. 1.234

Fraction – exercises

Calculate using `Fraction`:

1. $1/2 + 2/3$ `F(1,2) + F(2,3)`
2. $14\frac{4}{7} \cdot 11\frac{2}{19}$ `(14+F(4,7)) * (11+F(2,19))`
3. 0.8^3 `F('0.8') ** 3` or `F(8,10) ** 3` are better than `F(0.8) ** 3`

Approximate the following numbers by the fractions whose denominators are at most 81 (hint: `Fraction` has the `limit_denominator` method):

1. $55/703$
2. $3\frac{71}{501}$
3. π
4. 1.234

Fraction – exercises

Calculate using `Fraction`:

1. $1/2 + 2/3$ `F(1,2) + F(2,3)`
2. $14\frac{4}{7} \cdot 11\frac{2}{19}$ `(14+F(4,7)) * (11+F(2,19))`
3. 0.8^3 `F('0.8') ** 3` or `F(8,10) ** 3` are better than `F(0.8) ** 3`

Approximate the following numbers by the fractions whose denominators are at most 81 (hint: `Fraction` has the `limit_denominator` method):

1. $55/703$ `F(55,703).limit_denominator(81)`
2. $3\frac{71}{501}$ `(3+F(71,501)).limit_denominator(81)`
3. π `F(m.pi).limit_denominator(81)`
4. 1.234 `F('1.234').limit_denominator(81)` or `F(1234,1000).limit_denominator(81)` or `F(1.234).limit_denominator(81)`

Homework

Calculate:

1. a 2.5% tax on a \$1.40 charge (rounded to the nearest cent);
2. remainder after division of 10.5 by 0.2;
3. 500 significant decimal figures of Euler's constant;
4. $2.5 \cdot \frac{3}{5} + 2\frac{1}{10} \cdot 9\frac{2}{17} - 7.81$;
5. $c(a+b) + ab - 2c + \frac{a}{c}$, where $a = 1.1$, $b = -\frac{1}{2}$, $c = 3\frac{2}{21}$;
6. the approximation of Euler's constant by the fraction which denominator is at most 19;
7. $232_4/211_3 - 5463_8/325_6$; note that numbers are expressed in a few different numeral systems (which use different bases).

- Use the proper data types to get possibly accurate results.
- Please note all the expressions you used.

Bibliography

- Official *Python documentation* available on https://docs.python.org/3/, topics: *Decimal Floating Point Arithmetic* (in *The Python Tutorial*), modules: *decimal*, *fractions*
- Doug Hellmann *Python 3 Module of the Week* series, available on https://pymotw.com/3/, articles: *decimal – Fixed and floating point math*, *fractions – Rational Numbers*, *math – Mathematical Functions*
- Michael Borgwardt *What Every Programmer Should Know About Floating-Point Arithmetic*, available on http://floating-point-gui.de/
- *Floating point* in *Wikipedia, the free encyclopedia*, available on https://en.wikipedia.org/wiki/Floating_point