Python (3.x) shell as a calculator

basics of Python, its shell and mathematical modules

Piotr Beling

Uniwersytet Łódzki
(University of Łódź)

2016

Basis of ipython – exercise

1. Run *ipython qtconsole*:
   - Windows with Anaconda: Start → (All) Applications → Anaconda3 → Jupyter QTConsole
   - Linux: `ipython3 qtconsole`
2. Type: `s = "I␣like␣Matlab."` (and accept with `Enter` key)
   Now `s` is a variable which refers to the string `"I␣like␣Matlab."`.
3. Type: `s.` (notice a dot after `s`) and press `Tab` key.
   You should see a list of methods available for `s` (string).
4. Choose or type `replace` and open bracket:
   `s.replace(`
   You should see documentation (so-called *docstring*) for `replace` method of string.
5. Finish by putting arguments and pressing `Enter`:
   `s.replace("Matlab", "Python")`
6. Type and accept by `Enter`: `_ * 5`
   Underscore (_) refers to the output of the last statement.

Basis of ipython – exercise

1. Run *ipython qtconsole*:
   - Windows with Anaconda: Start → (All) Applications → Anaconda3 → Jupyter QTConsole
   - Linux: `ipython3 qtconsole`
2. Type: `s = "I␣like␣Matlab."` (and accept with `Enter` key)
   Now `s` is a variable which refers to the string `"I␣like␣Matlab."`.
3. Type: `s.` (notice a dot after `s`) and press `Tab` key.
   You should see a list of methods available for `s` (string).
4. Choose or type `replace` and open bracket:
   `s.replace(`
   You should see documentation (so-called *docstring*) for `replace` method of string.
5. Finish by putting arguments and pressing `Enter`:
   `s.replace("Matlab", "Python")`
6. Type and accept by `Enter`: `_ * 5`
   Underscore (_) refers to the output of the last statement.

Basis of ipython – exercise

1. Run *ipython qtconsole*:
   - Windows with Anaconda: Start → (All) Applications → Anaconda3 → Jupyter QTConsole
   - Linux: `ipython3 qtconsole`
2. Type: `s = "I␣like␣Matlab."` (and accept with `Enter` key)
   Now `s` is a variable which refers to the string `"I␣like␣Matlab."`.
3. Type: `s.` (notice a dot after `s`) and press `Tab` key.
   You should see a list of methods available for `s` (string).
4. Choose or type `replace` and open bracket:
   `s.replace(`
   You should see documentation (so-called *docstring*) for `replace` method of string.
5. Finish by putting arguments and pressing `Enter`:
   `s.replace("Matlab", "Python")`
6. Type and accept by `Enter`: `_ * 5`
   Underscore (_) refers to the output of the last statement.

Basis of ipython – exercise

1. Run *ipython qtconsole*:
   - Windows with Anaconda: Start → (All) Applications → Anaconda3 → Jupyter QTConsole
   - Linux: `ipython3 qtconsole`
2. Type: `s = "I␣like␣Matlab."` (and accept with `Enter` key)
   Now `s` is a variable which refers to the string `"I␣like␣Matlab."`.
3. Type: `s.` (notice a dot after `s`) and press `Tab` key.
   You should see a list of methods available for `s` (string).
4. Choose or type `replace` and open bracket:
   `s.replace(`
   You should see documentation (so-called *docstring*) for `replace` method of string.
5. Finish by putting arguments and pressing `Enter`:
   `s.replace("Matlab", "Python")`
6. Type and accept by `Enter`: `_ * 5`
   Underscore (_) refers to the output of the last statement.

Basis of ipython – exercise

1. Run *ipython qtconsole*:
   - Windows with Anaconda: Start → (All) Applications → Anaconda3 → Jupyter QTConsole
   - Linux: `ipython3 qtconsole`
2. Type: `s = "I␣like␣Matlab."` (and accept with `Enter` key)
   Now `s` is a variable which refers to the string `"I␣like␣Matlab."`.
3. Type: `s.` (notice a dot after `s`) and press `Tab` key.
   You should see a list of methods available for `s` (string).
4. Choose or type `replace` and open bracket:
   `s.replace(`
   You should see documentation (so-called *docstring*) for `replace` method of string.
5. Finish by putting arguments and pressing `Enter`:
   `s.replace("Matlab", "Python")`
6. Type and accept by `Enter`: `_ * 5`
   Underscore (_) refers to the output of the last statement.

Basis of ipython – exercise

1. Run *ipython qtconsole*:
   - Windows with Anaconda: Start → (All) Applications → Anaconda3 → Jupyter QTConsole
   - Linux: `ipython3 qtconsole`
2. Type: `s = "I␣like␣Matlab."` (and accept with `Enter` key)
   Now `s` is a variable which refers to the string `"I␣like␣Matlab."`.
3. Type: `s.` (notice a dot after `s`) and press `Tab` key.
   You should see a list of methods available for `s` (string).
4. Choose or type `replace` and open bracket:
   `s.replace(`
   You should see documentation (so-called *docstring*) for `replace` method of string.
5. Finish by putting arguments and pressing `Enter`:
   `s.replace("Matlab", "Python")`
6. Type and accept by `Enter`: `_ * 5`
   Underscore (_) refers to the output of the last statement.

Basis of ipython – command history

- One can refer to the outputs of previous statements by:
  - `_` (one underscore) – previous output,
  - `__` (two underscores) – next previous,
  - `___` (three underscores) – next-next previous,
  - `_<n>` (e.g. `_2`) – output of the statement number `n`.
    (Numbers of statements are displayed in brackets `[]`.)
- Up and down arrows can be used for navigation over commands.
  Exercise: find `_ * 5` in history and execute it again.
- You can also start typing, and then use arrows to search through only the history items that match what you have typed so far.
  Exercise: type `s` and use up and down arrows.

Basis of ipython – command history

- One can refer to the outputs of previous statements by:
  - `_` (one underscore) – previous output,
  - `__` (two underscores) – next previous,
  - `___` (three underscores) – next-next previous,
  - `_<n>` (e.g. `_2`) – output of the statement number `n`. (Numbers of statements are displayed in brackets `[]`.)
- Up and down arrows can be used for navigation over commands.

  Exercise: find `_ * 5` in history and execute it again.
- You can also start typing, and then use arrows to search through only the history items that match what you have typed so far.

  Exercise: type `s` and use up and down arrows.

Basis of ipython – command history

- One can refer to the outputs of previous statements by:
  - `_` (one underscore) – previous output,
  - `__` (two underscores) – next previous,
  - `___` (three underscores) – next-next previous,
  - `_<n>` (e.g. `_2`) – output of the statement number `n`. (Numbers of statements are displayed in brackets `[]`.)
- Up and down arrows can be used for navigation over commands.

  Exercise: find `_ * 5` in history and execute it again.
- You can also start typing, and then use arrows to search through only the history items that match what you have typed so far.

  Exercise: type `s` and use up and down arrows.

Basis of ipython – how to get help? [1/2]

- Typing `?sth`, `??sth`, `sth?` or `sth??` prints detailed information about an object, method or function `sth`.
  Examples:
  ```
  s?
  s.replace?
  ```
  Note that in case of using single question mark (`?`), very long docstrings are snipped.
- Astrix (∗) can be used to construct pattern and find names which match to it.
  For instance `?s.*find*` lists names in `s` containing `find`.

Exercises:

1. Display help about `find` method of `s`.
2. List names in `s` beginning with **`is`**.

Basis of ipython – how to get help? [1/2]

- Typing `?sth`, `??sth`, `sth?` or `sth??` prints detailed information about an object, method or function `sth`.
  Examples:
  ```
  s?
  s.replace?
  ```
  Note that in case of using single question mark (`?`), very long docstrings are snipped.
- Astrix (`*`) can be used to construct pattern and find names which match to it.
  For instance `?s.*find*` lists names in `s` containing `find`.

Exercises:

1. Display help about `find` method of `s`.
2. List names in `s` beginning with **`is`**.

Basis of ipython – how to get help? [1/2]

- Typing `?sth`, `??sth`, `sth?` or `sth??` prints detailed information about an object, method or function `sth`.
  Examples:
  ```
  s?
  s.replace?
  ```
  Note that in case of using single question mark (`?`), very long docstrings are snipped.
- Astrix (`*`) can be used to construct pattern and find names which match to it.
  For instance `?s.*find*` lists names in `s` containing `find`.

Exercises:

1. Display help about `find` method of `s`.
2. List names in `s` beginning with **`is`**.

Basis of ipython – how to get help? [1/2]

- Typing `?sth`, `??sth`, `sth?` or `sth??` prints detailed information about an object, method or function `sth`.
  Examples:
  `s?`
  `s.replace?`
  Note that in case of using single question mark (`?`), very long docstrings are snipped.
- Astrix (`*`) can be used to construct pattern and find names which match to it.
  For instance `?s.*find*` lists names in `s` containing `find`.

Exercises:

1. Display help about `find` method of `s`. Answer: `s.find?`
2. List names in `s` beginning with **`is`**. Answer: `?s.`**`is`**`*`

Basis of ipython – how to get help? [2/2]

- `help(sth)` displays help about module, keyword, or topic `sth`. For instance `help('str')` or `help(str)` (quotation marks can be omitted for built-in or already imported things).
- `help()` runs interactive help.
- *Help* menu includes further information.

Exercises:

1. Display help about `int`.
2. Display help about `sum` function.
3. Run interactive help and read welcoming message.
   Find all modules whose name or summary contains *math*.
   Finally, return to the interpreter.

Basis of ipython – how to get help? [2/2]

- **help**(sth) displays help about module, keyword, or topic sth. For instance **help**('str') or **help**(**str**) (quotation marks can be omitted for built-in or already imported things).
- **help**() runs interactive help.
- *Help* menu includes further information.

Exercises:

1. Display help about **int**.
2. Display help about **sum** function.
3. Run interactive help and read welcoming message.
   Find all modules whose name or summary contains *math*.
   Finally, return to the interpreter.

Basis of ipython – how to get help? [2/2]

- `help(sth)` displays help about module, keyword, or topic `sth`. For instance `help('str')` or `help(str)` (quotation marks can be omitted for built-in or already imported things).
- `help()` runs interactive help.
- *Help* menu includes further information.

Exercises:

1. Display help about `int`.
2. Display help about `sum` function.
3. Run interactive help and read welcoming message.
   Find all modules whose name or summary contains *math*.
   Finally, return to the interpreter.

Basis of ipython – how to get help? [2/2]

- **help**(sth) displays help about module, keyword, or topic sth. For instance **help**('str') or **help**(**str**) (quotation marks can be omitted for built-in or already imported things).
- **help**() runs interactive help.
- *Help* menu includes further information.

Exercises:

1. Display help about **int**.
2. Display help about **sum** function.
3. Run interactive help and read welcoming message.
   Find all modules whose name or summary contains *math*.
   Finally, return to the interpreter.

Basis of ipython – how to get help? [2/2]

- `help(sth)` displays help about module, keyword, or topic `sth`. For instance `help('str')` or `help(str)` (quotation marks can be omitted for built-in or already imported things).
- `help()` runs interactive help.
- *Help* menu includes further information.

Exercises:

1. Display help about `int`. Answer: `help('int')`, `help(int)` or `int?` (only about constructor)
2. Display help about `sum` function.
   Answer: `help('sum')`, `help(sum)` or `sum?`
3. Run interactive help and read welcoming message.
   Find all modules whose name or summary contains *math*.
   Finally, return to the interpreter.
   `help()`
   `modules math`
   `quit` (or just hit `Enter` without typing anything)

Basis of ipython – quitting, magic commands, ...

- To display a variable just enter its name or use the `print` function, e.g. type `s` or `print(s)`, and hit `Enter`.
- Shell can be closed (but do not do it now!) by executing: `quit`, `quit()`, `exit` or `exit()`, or pressing `ctrl+d`.
- Ipython supports so-called *magic* commands. Their names start with `%` (percent character).
- Magic commands can be accessed by typing their names (`Tab` key completes them) or by *Magic* menu.
- `%magic` print information about the magic function system. Please execute it now.
- `%time sth` and `%timeit sth` time execution of `sth` (and are examples of magic commands).
  Exercise: execute and compare the outputs of:
  ```
  %time s * 5
  %timeit s * 5
  ```

Basis of ipython – quitting, magic commands, ...

- To display a variable just enter its name or use the `print` function, e.g. type `s` or `print(s)`, and hit `Enter`.
- Shell can be closed (but do not do it now!) by executing: `quit`, `quit()`, `exit` or `exit()`, or pressing `ctrl+d`.
- Ipython supports so-called *magic* commands. Their names start with `%` (percent character).
- Magic commands can be accessed by typing their names (`Tab` key completes them) or by *Magic* menu.
- `%magic` print information about the magic function system. Please execute it now.
- `%time sth` and `%timeit sth` time execution of `sth` (and are examples of magic commands).
  Exercise: execute and compare the outputs of:

```
%time s * 5
%timeit s * 5
```

Basis of ipython – quitting, magic commands, ...

- To display a variable just enter its name or use the **`print`** function, e.g. type `s` or **`print`**`(s)`, and hit `Enter`.
- Shell can be closed (but do not do it now!) by executing: `quit`, `quit()`, `exit` or `exit()`, or pressing `ctrl+d`.
- Ipython supports so-called *magic* commands. Their names start with `%` (percent character).
- Magic commands can be accessed by typing their names (`Tab` key completes them) or by *Magic* menu.
- `%magic` print information about the magic function system. Please execute it now.
- `%time sth` and `%timeit sth` time execution of `sth` (and are examples of magic commands).

  Exercise: execute and compare the outputs of:

  ```
  %time s * 5
  %timeit s * 5
  ```

Basis of ipython – quitting, magic commands, ...

- To display a variable just enter its name or use the **`print`** function, e.g. type `s` or **`print`**`(s)`, and hit `Enter`.
- Shell can be closed (but do not do it now!) by executing: `quit`, `quit()`, `exit` or `exit()`, or pressing `ctrl+d`.
- Ipython supports so-called *magic* commands. Their names start with `%` (percent character).
- Magic commands can be accessed by typing their names (`Tab` key completes them) or by *Magic* menu.
- `%magic` print information about the magic function system. Please execute it now.
- `%time sth` and `%timeit sth` time execution of `sth` (and are examples of magic commands).
  Exercise: execute and compare the outputs of:
  `%time s * 5`
  `%timeit s * 5`

Basis of ipython – quitting, magic commands, ...

- To display a variable just enter its name or use the **print** function, e.g. type `s` or `print(s)`, and hit `Enter`.
- Shell can be closed (but do not do it now!) by executing: `quit`, `quit()`, `exit` or `exit()`, or pressing `ctrl+d`.
- Ipython supports so-called *magic* commands. Their names start with `%` (percent character).
- Magic commands can be accessed by typing their names (`Tab` key completes them) or by *Magic* menu.
- `%magic` print information about the magic function system. Please execute it now.
- `%time sth` and `%timeit sth` time execution of `sth` (and are examples of magic commands).

  Exercise: execute and compare the outputs of:

  `%time s * 5`

  `%timeit s * 5`

Basis of ipython – quitting, magic commands, ...

- To display a variable just enter its name or use the `print` function, e.g. type `s` or `print(s)`, and hit `Enter`.
- Shell can be closed (but do not do it now!) by executing: `quit`, `quit()`, `exit` or `exit()`, or pressing `ctrl+d`.
- Ipython supports so-called *magic* commands. Their names start with `%` (percent character).
- Magic commands can be accessed by typing their names (`Tab` key completes them) or by *Magic* menu.
- `%magic` print information about the magic function system. Please execute it now.
- `%time sth` and `%timeit sth` time execution of `sth` (and are examples of magic commands).

  Exercise: execute and compare the outputs of:

  ```
  %time s * 5
  %timeit s * 5
  ```

Variables and their types in Python

- Variables refer to (are labels for) objects in memory.
- For instance, at the moment, `s` refers to the object of the type **`str`** (string) which has a value `"I␣like␣Matlab."`. **Exercise:** execute `type(s)` to display the type of `s`.
- The same variables can be reused to store values of different types. **Exercise:**

```
s=1
type(s)
s=1.5
type(s)
```

- `del s` deletes the variable (label) `s`, but not object itself.
- All unreferenced objects are automatically deleted by a garbage collector. Automatic garbage collection is time-consuming and unpredictable, but it makes program development easier and less prone to error by relieving the developer of manual memory management.

Variables and their types in Python

- Variables refer to (are labels for) objects in memory.
- For instance, at the moment, `s` refers to the object of the type **`str`** (string) which has a value `"I␣like␣Matlab."`.
  Exercise: execute `type(s)` to display the type of `s`.
- The same variables can be reused to store values of different types. **Exercise:**

```
s=1
type(s)
s=1.5
type(s)
```

- `del s` deletes the variable (label) `s`, but not object itself.
- All unreferenced objects are automatically deleted by a garbage collector. Automatic garbage collection is time-consuming and unpredictable, but it makes program development easier and less prone to error by relieving the developer of manual memory management.

Variables and their types in Python

- Variables refer to (are labels for) objects in memory.
- For instance, at the moment, `s` refers to the object of the type **`str`** (string) which has a value `"I␣like␣Matlab."`.
  Exercise: execute `type(s)` to display the type of `s`.
- The same variables can be reused to store values of different types. **Exercise:**

```
s=1
type(s)
s=1.5
type(s)
```

- `del s` deletes the variable (label) `s`, but not object itself.
- All unreferenced objects are automatically deleted by a garbage collector. Automatic garbage collection is time-consuming and unpredictable, but it makes program development easier and less prone to error by relieving the developer of manual memory management.

Variables and their types in Python

- Variables refer to (are labels for) objects in memory.
- For instance, at the moment, `s` refers to the object of the type **`str`** (string) which has a value `"I␣like␣Matlab."`.
  Exercise: execute `type(s)` to display the type of `s`.
- The same variables can be reused to store values of different types. **Exercise:**

```
s=1
type(s)
s=1.5
type(s)
```

- `del s` deletes the variable (label) `s`, but not object itself.
- All unreferenced objects are automatically deleted by a garbage collector. Automatic garbage collection is time-consuming and unpredictable, but it makes program development easier and less prone to error by relieving the developer of manual memory management.

Variables and their types in Python

- Variables refer to (are labels for) objects in memory.
- For instance, at the moment, `s` refers to the object of the type **`str`** (string) which has a value `"I␣like␣Matlab."`.
  Exercise: execute `type(s)` to display the type of `s`.
- The same variables can be reused to store values of different types. **Exercise:**

```
s=1
type(s)
s=1.5
type(s)
```

- `del s` deletes the variable (label) `s`, but not object itself.
- All unreferenced objects are automatically deleted by a garbage collector. Automatic garbage collection is time-consuming and unpredictable, but it makes program development easier and less prone to error by relieving the developer of manual memory management.

Python shell as a calculator – examples / exercises [1/3]

Execute the following expressions:

`2 + 2`	simple sum of two integers (of `int` type);
`2 + 2*2`	Python follows the same precedence rules for its mathematical operators that mathematics does;
`(2+2)*2`	round brackets force a desired precedence;
`5.6 - 2`	dot (`.`) is used as a decimal separator; most operators convert numeric arguments to a common type, and the result is of that type (that is why `float` minus `int` gives `float`);
`2e18 * 5`	`2e18` is a `float` in scientific notation, equals $2 \cdot 10^{18}$;
`0.1+0.2`	`float` arithmetic is often not exact.

Calculate:

1. $5 - 3 \cdot 7$
2. $3.2 + 2.8$
3. $8 \cdot 4 \cdot (5.1 - 2.7)$
4. $0.1 + 0.7$

Python shell as a calculator – examples / exercises [1/3]

Execute the following expressions:

`2 + 2`	simple sum of two integers (of **`int`** type);
`2 + 2*2`	Python follows the same precedence rules for its mathematical operators that mathematics does;
`(2+2)*2`	round brackets force a desired precedence;
`5.6 - 2`	dot (`.`) is used as a decimal separator; most operators convert numeric arguments to a common type, and the result is of that type (that is why **`float`** minus **`int`** gives **`float`**);
`2e18 * 5`	`2e18` is a **`float`** in scientific notation, equals $2 \cdot 10^{18}$;
`0.1+0.2`	**`float`** arithmetic is often not exact.

Calculate:

1. $5 - 3 \cdot 7$
2. $3.2 + 2.8$
3. $8 \cdot 4 \cdot (5.1 - 2.7)$
4. $0.1 + 0.7$

Python shell as a calculator – examples / exercises [1/3]

Execute the following expressions:

`2 + 2`	simple sum of two integers (of **`int`** type);
`2 + 2*2`	Python follows the same precedence rules for its mathematical operators that mathematics does;
`(2+2)*2`	round brackets force a desired precedence;
`5.6 - 2`	dot (`.`) is used as a decimal separator; most operators convert numeric arguments to a common type, and the result is of that type (that is why **`float`** minus **`int`** gives **`float`**);
`2e18 * 5`	`2e18` is a **`float`** in scientific notation, equals $2 \cdot 10^{18}$;
`0.1+0.2`	**`float`** arithmetic is often not exact.

Calculate:

1. $5 - 3 \cdot 7$
2. $3.2 + 2.8$
3. $8 \cdot 4 \cdot (5.1 - 2.7)$
4. $0.1 + 0.7$

Python shell as a calculator – examples / exercises [1/3]

Execute the following expressions:

`2 + 2`	simple sum of two integers (of **`int`** type);
`2 + 2*2`	Python follows the same precedence rules for its mathematical operators that mathematics does;
`(2+2)*2`	round brackets force a desired precedence;
`5.6 - 2`	dot (`.`) is used as a decimal separator; most operators convert numeric arguments to a common type, and the result is of that type (that is why **`float`** minus **`int`** gives **`float`**);
`2e18 * 5`	2e18 is a **`float`** in scientific notation, equals $2 \cdot 10^{18}$;
`0.1+0.2`	**`float`** arithmetic is often not exact.

Calculate:

1. $5 - 3 \cdot 7$
2. $3.2 + 2.8$
3. $8 \cdot 4 \cdot (5.1 - 2.7)$
4. $0.1 + 0.7$

Python shell as a calculator – examples / exercises [1/3]

Execute the following expressions:

`2 + 2`	simple sum of two integers (of **`int`** type);
`2 + 2*2`	Python follows the same precedence rules for its mathematical operators that mathematics does;
`(2+2)*2`	round brackets force a desired precedence;
`5.6 - 2`	dot (`.`) is used as a decimal separator; most operators convert numeric arguments to a common type, and the result is of that type (that is why **`float`** minus **`int`** gives **`float`**);
`2e18 * 5`	`2e18` is a **`float`** in scientific notation, equals $2 \cdot 10^{18}$;
`0.1+0.2`	**`float`** arithmetic is often not exact.

Calculate:

1. $5 - 3 \cdot 7$
2. $3.2 + 2.8$
3. $8 \cdot 4 \cdot (5.1 - 2.7)$
4. $0.1 + 0.7$

Python shell as a calculator – examples / exercises [1/3]

Execute the following expressions:

`2 + 2`	simple sum of two integers (of **`int`** type);
`2 + 2*2`	Python follows the same precedence rules for its mathematical operators that mathematics does;
`(2+2)*2`	round brackets force a desired precedence;
`5.6 - 2`	dot (`.`) is used as a decimal separator; most operators convert numeric arguments to a common type, and the result is of that type (that is why **`float`** minus **`int`** gives **`float`**);
`2e18 * 5`	`2e18` is a **`float`** in scientific notation, equals $2 \cdot 10^{18}$;
`0.1+0.2`	**`float`** arithmetic is often not exact.

Calculate:

1. $5 - 3 \cdot 7$
2. $3.2 + 2.8$
3. $8 \cdot 4 \cdot (5.1 - 2.7)$
4. $0.1 + 0.7$

Python shell as a calculator – examples / exercises [1/3]

Execute the following expressions:

`2 + 2`	simple sum of two integers (of **`int`** type);
`2 + 2*2`	Python follows the same precedence rules for its mathematical operators that mathematics does;
`(2+2)*2`	round brackets force a desired precedence;
`5.6 - 2`	dot (`.`) is used as a decimal separator; most operators convert numeric arguments to a common type, and the result is of that type (that is why **`float`** minus **`int`** gives **`float`**);
`2e18 * 5`	`2e18` is a **`float`** in scientific notation, equals $2 \cdot 10^{18}$;
`0.1+0.2`	**`float`** arithmetic is often not exact.

Calculate:

1. $5 - 3 \cdot 7$
2. $3.2 + 2.8$
3. $8 \cdot 4 \cdot (5.1 - 2.7)$
4. $0.1 + 0.7$

Python shell as a calculator – examples / exercises [1/3]

Execute the following expressions:

`2 + 2`	simple sum of two integers (of **`int`** type);
`2 + 2*2`	Python follows the same precedence rules for its mathematical operators that mathematics does;
`(2+2)*2`	round brackets force a desired precedence;
`5.6 - 2`	dot (`.`) is used as a decimal separator; most operators convert numeric arguments to a common type, and the result is of that type (that is why **`float`** minus **`int`** gives **`float`**);
`2e18 * 5`	`2e18` is a **`float`** in scientific notation, equals $2 \cdot 10^{18}$;
`0.1+0.2`	**`float`** arithmetic is often not exact.

Calculate:

1. $5 - 3 \cdot 7$ Code: `5 - 3`
2. $3.2 + 2.8$ Code: `3.2 + 2.8`
3. $8 \cdot 4 \cdot (5.1 - 2.7)$ Code: `8 * 4 * (5.1-2.7)`
4. $0.1 + 0.7$ Code: `0.1+0.7`

Python shell as a calculator – examples / exercises [2/3]

Execute the following expressions:

`5 / 3`	normal division; **`int`** divided by **`int`** gives **`float`**;
`5 / 0`	division by 0 raises `ZeroDivisionError` exception;
`5 // 3`	$\lfloor \frac{5}{3} \rfloor$; floor division (//) gives **`int`** for **`int`**s operands;
`5 % 3`	remainder from the division of 5 by 3 (modulo operation); also **`int`** for **`int`**s operands;
`7.2 // 3`	for **`float`** and **`int`**, floor division gives integer encoded in **`float`** type;
`7.2 % 3`	also **`float`**, 1.2 since $3 \cdot \lfloor 7.2/3 \rfloor + 1.2 = 7.2$;

Calculate:

1. $1/3 + 0.1$
2. $\frac{2.7+4}{2} - 4$
3. $\lfloor 11.7/3.5 \rfloor$
4. $1/10 + 2/10$
5. $5 - 2 \cdot \lfloor 5/2 \rfloor$

Python shell as a calculator – examples / exercises [2/3]

Execute the following expressions:

`5 / 3`	normal division; **`int`** divided by **`int`** gives **`float`**;
`5 / 0`	division by 0 raises `ZeroDivisionError` exception;
`5 // 3`	$\lfloor \frac{5}{3} \rfloor$; floor division (//) gives **`int`** for **`int`**s operands;
`5 % 3`	remainder from the division of 5 by 3 (modulo operation); also **`int`** for **`int`**s operands;
`7.2 // 3`	for **`float`** and **`int`**, floor division gives integer encoded in **`float`** type;
`7.2 % 3`	also **`float`**, 1.2 since $3 \cdot \lfloor 7.2/3 \rfloor + 1.2 = 7.2$;

Calculate:

1. $1/3 + 0.1$
2. $\frac{2.7+4}{2} - 4$
3. $\lfloor 11.7/3.5 \rfloor$
4. $1/10 + 2/10$
5. $5 - 2 \cdot \lfloor 5/2 \rfloor$

Python shell as a calculator – examples / exercises [2/3]

Execute the following expressions:

`5 / 3`	normal division; `int` divided by `int` gives `float`;
`5 / 0`	division by 0 raises `ZeroDivisionError` exception;
`5 // 3`	$\lfloor \frac{5}{3} \rfloor$; floor division (`//`) gives `int` for `int`s operands;
`5 % 3`	remainder from the division of 5 by 3 (modulo operation); also `int` for `int`s operands;
`7.2 // 3`	for `float` and `int`, floor division gives integer encoded in `float` type;
`7.2 % 3`	also `float`, 1.2 since $3 \cdot \lfloor 7.2/3 \rfloor + 1.2 = 7.2$;

Calculate:

1. $1/3 + 0.1$
2. $\frac{2.7+4}{2} - 4$
3. $\lfloor 11.7/3.5 \rfloor$
4. $1/10 + 2/10$
5. $5 - 2 \cdot \lfloor 5/2 \rfloor$

Python shell as a calculator – examples / exercises [2/3]

Execute the following expressions:

`5 / 3`	normal division; **`int`** divided by **`int`** gives **`float`**;
`5 / 0`	division by 0 raises `ZeroDivisionError` exception;
`5 // 3`	$\lfloor \frac{5}{3} \rfloor$; floor division (`//`) gives **`int`** for **`int`**s operands;
`5 % 3`	remainder from the division of 5 by 3 (modulo operation); also **`int`** for **`int`**s operands;
`7.2 // 3`	for **`float`** and **`int`**, floor division gives integer encoded in **`float`** type;
`7.2 % 3`	also **`float`**, 1.2 since $3 \cdot \lfloor 7.2/3 \rfloor + 1.2 = 7.2$;

Calculate:

1. $1/3 + 0.1$
2. $\frac{2.7+4}{2} - 4$
3. $\lfloor 11.7/3.5 \rfloor$
4. $1/10 + 2/10$
5. $5 - 2 \cdot \lfloor 5/2 \rfloor$

Python shell as a calculator – examples / exercises [2/3]

Execute the following expressions:

`5 / 3`	normal division; **`int`** divided by **`int`** gives **`float`**;
`5 / 0`	division by 0 raises `ZeroDivisionError` exception;
`5 // 3`	$\lfloor \frac{5}{3} \rfloor$; floor division (`//`) gives **`int`** for **`int`**s operands;
`5 % 3`	remainder from the division of 5 by 3 (modulo operation); also **`int`** for **`int`**s operands;
`7.2 // 3`	for **`float`** and **`int`**, floor division gives integer encoded in **`float`** type;
`7.2 % 3`	also **`float`**, 1.2 since $3 \cdot \lfloor 7.2/3 \rfloor + 1.2 = 7.2$;

Calculate:

1. $1/3 + 0.1$
2. $\frac{2.7+4}{2} - 4$
3. $\lfloor 11.7/3.5 \rfloor$
4. $1/10 + 2/10$
5. $5 - 2 \cdot \lfloor 5/2 \rfloor$

Python shell as a calculator – examples / exercises [2/3]

Execute the following expressions:

`5 / 3`	normal division; **`int`** divided by **`int`** gives **`float`**;
`5 / 0`	division by 0 raises `ZeroDivisionError` exception;
`5 // 3`	$\lfloor \frac{5}{3} \rfloor$; floor division (`//`) gives **`int`** for **`int`**s operands;
`5 % 3`	remainder from the division of 5 by 3 (modulo operation); also **`int`** for **`int`**s operands;
`7.2 // 3`	for **`float`** and **`int`**, floor division gives integer encoded in **`float`** type;
`7.2 % 3`	also **`float`**, 1.2 since $3 \cdot \lfloor 7.2/3 \rfloor + 1.2 = 7.2$;

Calculate:

1. $1/3 + 0.1$
2. $\frac{2.7+4}{2} - 4$
3. $\lfloor 11.7/3.5 \rfloor$
4. $1/10 + 2/10$
5. $5 - 2 \cdot \lfloor 5/2 \rfloor$

Python shell as a calculator – examples / exercises [2/3]

Execute the following expressions:

`5 / 3`	normal division; **`int`** divided by **`int`** gives **`float`**;
`5 / 0`	division by 0 raises `ZeroDivisionError` exception;
`5 // 3`	$\lfloor \frac{5}{3} \rfloor$; floor division (`//`) gives **`int`** for **`int`**s operands;
`5 % 3`	remainder from the division of 5 by 3 (modulo operation); also **`int`** for **`int`**s operands;
`7.2 // 3`	for **`float`** and **`int`**, floor division gives integer encoded in **`float`** type;
`7.2 % 3`	also **`float`**, 1.2 since $3 \cdot \lfloor 7.2/3 \rfloor + 1.2 = 7.2$;

Calculate:

1. $1/3 + 0.1$
2. $\frac{2.7+4}{2} - 4$
3. $\lfloor 11.7/3.5 \rfloor$
4. $1/10 + 2/10$
5. $5 - 2 \cdot \lfloor 5/2 \rfloor$

Python shell as a calculator – examples / exercises [2/3]

Execute the following expressions:

`5 / 3`	normal division; **`int`** divided by **`int`** gives **`float`**;
`5 / 0`	division by 0 raises `ZeroDivisionError` exception;
`5 // 3`	$\lfloor \frac{5}{3} \rfloor$; floor division (`//`) gives **`int`** for **`int`**s operands;
`5 % 3`	remainder from the division of 5 by 3 (modulo operation); also **`int`** for **`int`**s operands;
`7.2 // 3`	for **`float`** and **`int`**, floor division gives integer encoded in **`float`** type;
`7.2 % 3`	also **`float`**, 1.2 since $3 \cdot \lfloor 7.2/3 \rfloor + 1.2 = 7.2$;

Calculate:

1. $1/3 + 0.1$ Code: `1/3 + 0.1`
2. $\frac{2.7+4}{2} - 4$ Code: `(2.7+4)/2 - 4`
3. $\lfloor 11.7/3.5 \rfloor$ Code: `11.7 // 3.5`
4. $1/10 + 2/10$ Code: `1/10 + 2/10`
5. $5 - 2 \cdot \lfloor 5/2 \rfloor$ Code: `5-2*(5//2)` or `5%2`

Python shell as a calculator – examples / exercises [3/3]

Execute the following expressions:

`7 ** 82`	7 to the power of 82 can be calculated precisely due to support for arbitrary-precision integers;
`pow(7,82)`	another notation for `7**82`;
`7.0 ** 82`	`float` is not of arbitrary-precision;
`2 ** (1/2)`	square root of 2 ($\sqrt{2}$); `float` since `1/2` is `float`;
`2 ** -3`	also negative exponent yields to `float` result; same as `1/(2**3)`;
`abs(-3.6)`	`abs` calculates absolute value and usually preserves the type of the argument.

Calculate:

1. $|-9^{53}|$
2. $\sqrt[3]{5}$
3. $2.1^{-5} + 1/3$
4. $|\frac{73+29}{32-76}|^{\lfloor 14/3 \rfloor}$

Python shell as a calculator – examples / exercises [3/3]

Execute the following expressions:

`7 ** 82`	7 to the power of 82 can be calculated precisely due to support for arbitrary-precision integers;
`pow(7,82)`	another notation for `7**82`;
`7.0 ** 82`	`float` is not of arbitrary-precision;
`2 ** (1/2)`	square root of 2 ($\sqrt{2}$); `float` since `1/2` is `float`;
`2 ** -3`	also negative exponent yields to `float` result; same as `1/(2**3)`;
`abs(-3.6)`	`abs` calculates absolute value and usually preserves the type of the argument.

Calculate:

1. $|-9^{53}|$
2. $\sqrt[3]{5}$
3. $2.1^{-5} + 1/3$
4. $\left|\frac{73+29}{32-76}\right|^{\lfloor 14/3 \rfloor}$

Python shell as a calculator – examples / exercises [3/3]

Execute the following expressions:

`7 ** 82`	7 to the power of 82 can be calculated precisely due to support for arbitrary-precision integers;
`pow(7,82)`	another notation for `7**82`;
`7.0 ** 82`	`float` is not of arbitrary-precision;
`2 ** (1/2)`	square root of 2 ($\sqrt{2}$); `float` since `1/2` is `float`;
`2 ** -3`	also negative exponent yields to `float` result; same as `1/(2**3)`;
`abs(-3.6)`	`abs` calculates absolute value and usually preserves the type of the argument.

Calculate:

1. $|-9^{53}|$
2. $\sqrt[3]{5}$
3. $2.1^{-5} + 1/3$
4. $|\frac{73+29}{32-76}|^{\lfloor 14/3 \rfloor}$

Python shell as a calculator – examples / exercises [3/3]

Execute the following expressions:

`7 ** 82`	7 to the power of 82 can be calculated precisely due to support for arbitrary-precision integers;
`pow(7,82)`	another notation for `7**82`;
`7.0 ** 82`	`float` is not of arbitrary-precision;
`2 ** (1/2)`	square root of 2 ($\sqrt{2}$); `float` since `1/2` is `float`;
`2 ** -3`	also negative exponent yields to `float` result; same as `1/(2**3)`;
`abs(-3.6)`	`abs` calculates absolute value and usually preserves the type of the argument.

Calculate:

1. $|-9^{53}|$
2. $\sqrt[3]{5}$
3. $2.1^{-5} + 1/3$
4. $\left|\frac{73+29}{32-76}\right|^{\lfloor 14/3 \rfloor}$

Python shell as a calculator – examples / exercises [3/3]

Execute the following expressions:

`7 ** 82`	7 to the power of 82 can be calculated precisely due to support for arbitrary-precision integers;
`pow(7,82)`	another notation for `7**82`;
`7.0 ** 82`	`float` is not of arbitrary-precision;
`2 ** (1/2)`	square root of 2 ($\sqrt{2}$); `float` since `1/2` is `float`;
`2 ** -3`	also negative exponent yields to `float` result; same as `1/(2**3)`;
`abs(-3.6)`	`abs` calculates absolute value and usually preserves the type of the argument.

Calculate:

1. $|-9^{53}|$
2. $\sqrt[3]{5}$
3. $2.1^{-5} + 1/3$
4. $\left|\frac{73+29}{32-76}\right|^{\lfloor 14/3 \rfloor}$

Python shell as a calculator – examples / exercises [3/3]

Execute the following expressions:

`7 ** 82`	7 to the power of 82 can be calculated precisely due to support for arbitrary-precision integers;
`pow(7,82)`	another notation for `7**82`;
`7.0 ** 82`	**`float`** is not of arbitrary-precision;
`2 ** (1/2)`	square root of 2 ($\sqrt{2}$); **`float`** since `1/2` is **`float`**;
`2 ** -3`	also negative exponent yields to **`float`** result; same as `1/(2**3)`;
`abs(-3.6)`	**`abs`** calculates absolute value and usually preserves the type of the argument.

Calculate:

1. $|-9^{53}|$
2. $\sqrt[3]{5}$
3. $2.1^{-5} + 1/3$
4. $\left|\frac{73+29}{32-76}\right|^{\lfloor 14/3 \rfloor}$

Python shell as a calculator – examples / exercises [3/3]

Execute the following expressions:

`7 ** 82`	7 to the power of 82 can be calculated precisely due to support for arbitrary-precision integers;
`pow(7,82)`	another notation for `7**82`;
`7.0 ** 82`	**`float`** is not of arbitrary-precision;
`2 ** (1/2)`	square root of 2 ($\sqrt{2}$); **`float`** since `1/2` is **`float`**;
`2 ** -3`	also negative exponent yields to **`float`** result; same as `1/(2**3)`;
`abs(-3.6)`	**`abs`** calculates absolute value and usually preserves the type of the argument.

Calculate:

1. $|-9^{53}|$
2. $\sqrt[3]{5}$
3. $2.1^{-5} + 1/3$
4. $|\frac{73+29}{32-76}|^{\lfloor 14/3 \rfloor}$

Python shell as a calculator – examples / exercises [3/3]

Execute the following expressions:

`7 ** 82`	7 to the power of 82 can be calculated precisely due to support for arbitrary-precision integers;
`pow(7,82)`	another notation for `7**82`;
`7.0 ** 82`	`float` is not of arbitrary-precision;
`2 ** (1/2)`	square root of 2 ($\sqrt{2}$); `float` since `1/2` is `float`;
`2 ** -3`	also negative exponent yields to `float` result; same as `1/(2**3)`;
`abs(-3.6)`	`abs` calculates absolute value and usually preserves the type of the argument.

Calculate:

1. $|-9^{53}|$ Code: `abs(-9 ** 53)`
2. $\sqrt[3]{5}$ Code: `5 ** (1/3)`
3. $2.1^{-5} + 1/3$ Code: `2.1**-5 + 1/3`
4. $\left|\frac{73+29}{32-76}\right|^{\lfloor 14/3 \rfloor}$ Code: `abs((73+29)/(32-76))**(14//3)`

Comparison (relational) operators

Examples:

`2 * 2 == 4`	is `True`;
`3 != 3`	is `False`;
`1 < 2`	is `True`;
`5 > 5`	is `False`;
`5 >= 5`	is `True`;
`0 < 3 < 5`	is `True`;
`5 >= 3 > 8`	is `False`;
`9 == 9 > 1`	is `True`;
`3 = 3`	throws `SyntaxError`: `can't assign to` literal (to `3`).

Exercise: check if $3^7 \geq 7^3 > 100$

Comparison operators:	
`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Do not confuse:

- assignment operator `=`
- with equality check `==`

Comparison (relational) operators

Examples:

`2 * 2 == 4`	is `True`;
`3 != 3`	is `False`;
`1 < 2`	is `True`;
`5 > 5`	is `False`;
`5 >= 5`	is `True`;
`0 < 3 < 5`	is `True`;
`5 >= 3 > 8`	is `False`;
`9 == 9 > 1`	is `True`;
`3 = 3`	throws `SyntaxError`: can't assign to literal (to `3`).

Exercise: check if $3^7 \geq 7^3 > 100$

Comparison operators:	
`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Do not confuse:

- assignment operator `=`
- with equality check `==`

Comparison (relational) operators

Examples:

`2 * 2 == 4`	is `True`;
`3 != 3`	is `False`;
`1 < 2`	is `True`;
`5 > 5`	is `False`;
`5 >= 5`	is `True`;
`0 < 3 < 5`	is `True`;
`5 >= 3 > 8`	is `False`;
`9 == 9 > 1`	is `True`;
`3 = 3`	throws `SyntaxError`: can't assign to literal (to `3`).

Comparison operators:

`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Do not confuse:

- assignment operator `=`
- with equality check `==`

Exercise: check if $3^7 \geq 7^3 > 100$

Comparison (relational) operators

Examples:

`2 * 2 == 4`	is `True`;
`3 != 3`	is `False`;
`1 < 2`	is `True`;
`5 > 5`	is `False`;
`5 >= 5`	is `True`;
`0 < 3 < 5`	is `True`;
`5 >= 3 > 8`	is `False`;
`9 == 9 > 1`	is `True`;
`3 = 3`	throws `SyntaxError`: can't assign to literal (to `3`).

Comparison operators:

`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Do not confuse:

- assignment operator `=`
- with equality check `==`

Exercise: check if $3^7 \geq 7^3 > 100$

Comparison (relational) operators

Examples:

`2 * 2 == 4`	is `True`;
`3 != 3`	is `False`;
`1 < 2`	is `True`;
`5 > 5`	is `False`;
`5 >= 5`	is `True`;
`0 < 3 < 5`	is `True`;
`5 >= 3 > 8`	is `False`;
`9 == 9 > 1`	is `True`;
`3 = 3`	throws `SyntaxError`: can't assign to literal (to `3`).

Comparison operators:

`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Do not confuse:

- assignment operator `=`
- with equality check `==`

Exercise: check if $3^7 \geq 7^3 > 100$

Comparison (relational) operators

Examples:

`2 * 2 == 4`	is `True`;
`3 != 3`	is `False`;
`1 < 2`	is `True`;
`5 > 5`	is `False`;
`5 >= 5`	is `True`;
`0 < 3 < 5`	is `True`;
`5 >= 3 > 8`	is `False`;
`9 == 9 > 1`	is `True`;
`3 = 3`	throws `SyntaxError`: can't assign to literal (to `3`).

Comparison operators:

`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Do not confuse:

- assignment operator `=`
- with equality check `==`

Exercise: check if $3^7 \geq 7^3 > 100$

Comparison (relational) operators

Examples:

`2 * 2 == 4`	is `True`;
`3 != 3`	is `False`;
`1 < 2`	is `True`;
`5 > 5`	is `False`;
`5 >= 5`	is `True`;
`0 < 3 < 5`	is `True`;
`5 >= 3 > 8`	is `False`;
`9 == 9 > 1`	is `True`;
`3 = 3`	throws `SyntaxError`: can't assign to literal (to `3`).

Exercise: check if $3^7 \geq 7^3 > 100$

Comparison operators:	
`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Do not confuse:

- assignment operator `=`
- with equality check `==`

Comparison (relational) operators

Examples:

`2 * 2 == 4`	is `True`;
`3 != 3`	is `False`;
`1 < 2`	is `True`;
`5 > 5`	is `False`;
`5 >= 5`	is `True`;
`0 < 3 < 5`	is `True`;
`5 >= 3 > 8`	is `False`;
`9 == 9 > 1`	is `True`;
`3 = 3`	throws `SyntaxError`: can't assign to literal (to `3`).

Comparison operators:	
`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Do not confuse:

- assignment operator `=`
- with equality check `==`

Exercise: check if $3^7 \geq 7^3 > 100$

Comparison (relational) operators

Examples:

`2 * 2 == 4`	is `True`;
`3 != 3`	is `False`;
`1 < 2`	is `True`;
`5 > 5`	is `False`;
`5 >= 5`	is `True`;
`0 < 3 < 5`	is `True`;
`5 >= 3 > 8`	is `False`;
`9 == 9 > 1`	is `True`;
`3 = 3`	throws `SyntaxError`: can't assign to literal (to `3`).

Exercise: check if $3^7 \geq 7^3 > 100$

Comparison operators:	
`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Do not confuse:

- assignment operator `=`
- with equality check `==`

Comparison (relational) operators

Examples:

`2 * 2 == 4`	is `True`;
`3 != 3`	is `False`;
`1 < 2`	is `True`;
`5 > 5`	is `False`;
`5 >= 5`	is `True`;
`0 < 3 < 5`	is `True`;
`5 >= 3 > 8`	is `False`;
`9 == 9 > 1`	is `True`;
`3 = 3`	throws `SyntaxError`: can't assign to literal (to `3`).

Exercise: check if $3^7 \geq 7^3 > 100$

Comparison operators:

`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Do not confuse:

- assignment operator `=`
- with equality check `==`

Comparison (relational) operators

Examples:

`2 * 2 == 4`	is `True`;
`3 != 3`	is `False`;
`1 < 2`	is `True`;
`5 > 5`	is `False`;
`5 >= 5`	is `True`;
`0 < 3 < 5`	is `True`;
`5 >= 3 > 8`	is `False`;
`9 == 9 > 1`	is `True`;
`3 = 3`	throws `SyntaxError`: can't assign to literal (to `3`).

Comparison operators:	
`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Exercise: check if $3^7 \geq 7^3 > 100$

Code: `3**7 >= 7**3 > 100`

Do not confuse:

- assignment operator `=`
- with equality check `==`

Complex numbers – examples / exercises

Python supports computation with complex numbers.

Execute the following expressions:

`a = 1+2j`	`j` represents the imaginary unit ($\sqrt{-1}$);
`type(a)`	type of `a` is `complex`;
`b = complex(3, 1)`	same as `b = 3+1j`;
`a + b`	sum of `complexes` is also `complex`;
`a + 2`	`complex` plus `int` gives `complex`;
`a.real`	real part is of the type `float`;
`a.imag`	imaginary part, also `float`.

Calculate:

1. $a^3 + 2/b$
2. j^2
3. $\frac{a+b}{2} - 5j$
4. $\mathrm{Re}(a - b) \cdot 3$, where $\mathrm{Re}(x)$ denotes a real part of x

Complex numbers – examples / exercises

Python supports computation with complex numbers.
Execute the following expressions:

`a = 1+2j`	`j` represents the imaginary unit ($\sqrt{-1}$);
`type(a)`	type of `a` is **`complex`**;
`b = complex(3, 1)`	same as `b = 3+1j`;
`a + b`	sum of **`complex`**es is also **`complex`**;
`a + 2`	**`complex`** plus **`int`** gives **`complex`**;
`a.real`	real part is of the type **`float`**;
`a.imag`	imaginary part, also **`float`**.

Calculate:

1. $a^3 + 2/b$
2. j^2
3. $\frac{a+b}{2} - 5j$
4. $\mathrm{Re}(a - b) \cdot 3$, where $\mathrm{Re}(x)$ denotes a real part of x

Complex numbers – examples / exercises

Python supports computation with complex numbers.
Execute the following expressions:

`a = 1+2j`	`j` represents the imaginary unit ($\sqrt{-1}$);
`type(a)`	type of `a` is **`complex`**;
`b = complex(3, 1)`	same as `b = 3+1j`;
`a + b`	sum of **`complex`**es is also **`complex`**;
`a + 2`	**`complex`** plus **`int`** gives **`complex`**;
`a.real`	real part is of the type **`float`**;
`a.imag`	imaginary part, also **`float`**.

Calculate:

1. $a^3 + 2/b$
2. j^2
3. $\frac{a+b}{2} - 5j$
4. $\mathrm{Re}(a - b) \cdot 3$, where $\mathrm{Re}(x)$ denotes a real part of x

Complex numbers – examples / exercises

Python supports computation with complex numbers.
Execute the following expressions:

`a = 1+2j`	`j` represents the imaginary unit ($\sqrt{-1}$);
`type(a)`	type of `a` is **`complex`**;
`b = complex(3, 1)`	same as `b = 3+1j`;
`a + b`	sum of **`complexes`** is also **`complex`**;
`a + 2`	**`complex`** plus **`int`** gives **`complex`**;
`a.real`	real part is of the type **`float`**;
`a.imag`	imaginary part, also **`float`**.

Calculate:

1. $a^3 + 2/b$
2. j^2
3. $\frac{a+b}{2} - 5j$
4. $\mathrm{Re}(a-b) \cdot 3$, where $\mathrm{Re}(x)$ denotes a real part of x

Complex numbers – examples / exercises

Python supports computation with complex numbers.
Execute the following expressions:

`a = 1+2j`	`j` represents the imaginary unit ($\sqrt{-1}$);
`type(a)`	type of `a` is **`complex`**;
`b = complex(3, 1)`	same as `b = 3+1j`;
`a + b`	sum of **`complex`**es is also **`complex`**;
`a + 2`	**`complex`** plus **`int`** gives **`complex`**;
`a.real`	real part is of the type **`float`**;
`a.imag`	imaginary part, also **`float`**.

Calculate:

1. $a^3 + 2/b$
2. j^2
3. $\frac{a+b}{2} - 5j$
4. $\mathrm{Re}(a - b) \cdot 3$, where $\mathrm{Re}(x)$ denotes a real part of x

Complex numbers – examples / exercises

Python supports computation with complex numbers.
Execute the following expressions:

`a = 1+2j`	`j` represents the imaginary unit ($\sqrt{-1}$);
`type(a)`	type of `a` is **`complex`**;
`b = complex(3, 1)`	same as `b = 3+1j`;
`a + b`	sum of **`complex`**es is also **`complex`**;
`a + 2`	**`complex`** plus **`int`** gives **`complex`**;
`a.real`	real part is of the type **`float`**;
`a.imag`	imaginary part, also **`float`**.

Calculate:

1. $a^3 + 2/b$
2. j^2
3. $\frac{a+b}{2} - 5j$
4. $\text{Re}(a - b) \cdot 3$, where $\text{Re}(x)$ denotes a real part of x

Complex numbers – examples / exercises

Python supports computation with complex numbers.
Execute the following expressions:

`a = 1+2j`	`j` represents the imaginary unit ($\sqrt{-1}$);
`type(a)`	type of `a` is **`complex`**;
`b = complex(3, 1)`	same as `b = 3+1j`;
`a + b`	sum of **`complex`**es is also **`complex`**;
`a + 2`	**`complex`** plus **`int`** gives **`complex`**;
`a.real`	real part is of the type **`float`**;
`a.imag`	imaginary part, also **`float`**.

Calculate:

1. $a^3 + 2/b$
2. j^2
3. $\frac{a+b}{2} - 5j$
4. $\mathrm{Re}(a - b) \cdot 3$, where $\mathrm{Re}(x)$ denotes a real part of x

Complex numbers – examples / exercises

Python supports computation with complex numbers.
Execute the following expressions:

`a = 1+2j`	`j` represents the imaginary unit ($\sqrt{-1}$);
`type(a)`	type of `a` is **`complex`**;
`b = complex(3, 1)`	same as `b = 3+1j`;
`a + b`	sum of **`complex`**es is also **`complex`**;
`a + 2`	**`complex`** plus **`int`** gives **`complex`**;
`a.real`	real part is of the type **`float`**;
`a.imag`	imaginary part, also **`float`**.

Calculate:

1. $a^3 + 2/b$
2. j^2
3. $\frac{a+b}{2} - 5j$
4. $\mathrm{Re}(a - b) \cdot 3$, where $\mathrm{Re}(x)$ denotes a real part of x

Complex numbers – examples / exercises

Python supports computation with complex numbers.
Execute the following expressions:

`a = 1+2j`	`j` represents the imaginary unit ($\sqrt{-1}$);
`type(a)`	type of `a` is **`complex`**;
`b = complex(3, 1)`	same as `b = 3+1j`;
`a + b`	sum of **`complex`**es is also **`complex`**;
`a + 2`	**`complex`** plus **`int`** gives **`complex`**;
`a.real`	real part is of the type **`float`**;
`a.imag`	imaginary part, also **`float`**.

Calculate:

1. $a^3 + 2/b$
2. j^2
3. $\frac{a+b}{2} - 5j$
4. $\mathrm{Re}(a - b) \cdot 3$, where $\mathrm{Re}(x)$ denotes a real part of x

Complex numbers – examples / exercises

Python supports computation with complex numbers.
Execute the following expressions:

`a = 1+2j`	`j` represents the imaginary unit ($\sqrt{-1}$);
`type(a)`	type of `a` is **`complex`**;
`b = complex(3, 1)`	same as `b = 3+1j`;
`a + b`	sum of **`complex`**es is also **`complex`**;
`a + 2`	**`complex`** plus **`int`** gives **`complex`**;
`a.real`	real part is of the type **`float`**;
`a.imag`	imaginary part, also **`float`**.

Calculate:

1. $a^3 + 2/b$ Code: `a**3 + 2/b`
2. j^2 Code: `1j**2`
3. $\frac{a+b}{2} - 5j$ Code: `(a+b)/2 - 5j`
4. $\mathrm{Re}(a - b) \cdot 3$, where $\mathrm{Re}(x)$ denotes a real part of x
   Code: `(a-b).real * 3`

Conversions between types

Execute the following expressions:

`str(57)`	`int` to `str` conversion;
`int(2.83)`	`float` to `int`; discards non-integer digits;
`round(2.83)`	`float` rounded to the nearest `int`;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (`float`) type;
`complex('1+2j')`	`str` to `complex` conversion;
`float('-2.6')`	`str` to `float` conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	`str` to `int` conversion;
`int('101', 2)`	binary (base 2) number as `str`, to `int`;
`bin(18)`	`int` to `str` in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`)

Conversions between types

Execute the following expressions:

`str(57)`	`int` to `str` conversion;
`int(2.83)`	`float` to `int`; discards non-integer digits;
`round(2.83)`	`float` rounded to the nearest `int`;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (`float`) type;
`complex('1+2j')`	`str` to `complex` conversion;
`float('-2.6')`	`str` to `float` conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	`str` to `int` conversion;
`int('101', 2)`	binary (base 2) number as `str`, to `int`;
`bin(18)`	`int` to `str` in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`)

Conversions between types

Execute the following expressions:

`str(57)`	`int` to `str` conversion;
`int(2.83)`	`float` to `int`; discards non-integer digits;
`round(2.83)`	`float` rounded to the nearest `int`;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (`float`) type;
`complex('1+2j')`	`str` to `complex` conversion;
`float('-2.6')`	`str` to `float` conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	`str` to `int` conversion;
`int('101', 2)`	binary (base 2) number as `str`, to `int`;
`bin(18)`	`int` to `str` in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`)

Conversions between types

Execute the following expressions:

`str(57)`	`int` to `str` conversion;
`int(2.83)`	`float` to `int`; discards non-integer digits;
`round(2.83)`	`float` rounded to the nearest `int`;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (`float`) type;
`complex('1+2j')`	`str` to `complex` conversion;
`float('-2.6')`	`str` to `float` conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	`str` to `int` conversion;
`int('101', 2)`	binary (base 2) number as `str`, to `int`;
`bin(18)`	`int` to `str` in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`)

Conversions between types

Execute the following expressions:

`str(57)`	`int` to `str` conversion;
`int(2.83)`	`float` to `int`; discards non-integer digits;
`round(2.83)`	`float` rounded to the nearest `int`;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (`float`) type;
`complex('1+2j')`	`str` to `complex` conversion;
`float('-2.6')`	`str` to `float` conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	`str` to `int` conversion;
`int('101', 2)`	binary (base 2) number as `str`, to `int`;
`bin(18)`	`int` to `str` in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`)

Conversions between types

Execute the following expressions:

`str(57)`	`int` to `str` conversion;
`int(2.83)`	`float` to `int`; discards non-integer digits;
`round(2.83)`	`float` rounded to the nearest `int`;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (`float`) type;
`complex('1+2j')`	`str` to `complex` conversion;
`float('-2.6')`	`str` to `float` conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	`str` to `int` conversion;
`int('101', 2)`	binary (base 2) number as `str`, to `int`;
`bin(18)`	`int` to `str` in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`)

Conversions between types

Execute the following expressions:

`str(57)`	`int` to `str` conversion;
`int(2.83)`	`float` to `int`; discards non-integer digits;
`round(2.83)`	`float` rounded to the nearest `int`;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (`float`) type;
`complex('1+2j')`	`str` to `complex` conversion;
`float('-2.6')`	`str` to `float` conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	`str` to `int` conversion;
`int('101', 2)`	binary (base 2) number as `str`, to `int`;
`bin(18)`	`int` to `str` in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`)

Conversions between types

Execute the following expressions:

`str(57)`	`int` to `str` conversion;
`int(2.83)`	`float` to `int`; discards non-integer digits;
`round(2.83)`	`float` rounded to the nearest `int`;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (`float`) type;
`complex('1+2j')`	`str` to `complex` conversion;
`float('-2.6')`	`str` to `float` conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	`str` to `int` conversion;
`int('101', 2)`	binary (base 2) number as `str`, to `int`;
`bin(18)`	`int` to `str` in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`)

Conversions between types

Execute the following expressions:

`str(57)`	`int` to `str` conversion;
`int(2.83)`	`float` to `int`; discards non-integer digits;
`round(2.83)`	`float` rounded to the nearest `int`;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (`float`) type;
`complex('1+2j')`	`str` to `complex` conversion;
`float('-2.6')`	`str` to `float` conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	`str` to `int` conversion;
`int('101', 2)`	binary (base 2) number as `str`, to `int`;
`bin(18)`	`int` to `str` in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`)

Conversions between types

Execute the following expressions:

`str(57)`	`int` to `str` conversion;
`int(2.83)`	`float` to `int`; discards non-integer digits;
`round(2.83)`	`float` rounded to the nearest `int`;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (`float`) type;
`complex('1+2j')`	`str` to `complex` conversion;
`float('-2.6')`	`str` to `float` conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	`str` to `int` conversion;
`int('101', 2)`	binary (base 2) number as `str`, to `int`;
`bin(18)`	`int` to `str` in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`)

Conversions between types

Execute the following expressions:

`str(57)`	**int** to **str** conversion;
`int(2.83)`	**float** to **int**; discards non-integer digits;
`round(2.83)`	**float** rounded to the nearest **int**;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (**float**) type;
`complex('1+2j')`	**str** to **complex** conversion;
`float('-2.6')`	**str** to **float** conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	**str** to **int** conversion;
`int('101', 2)`	binary (base 2) number as **str**, to **int**;
`bin(18)`	**int** to **str** in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`)

Conversions between types

Execute the following expressions:

`str(57)`	`int` to `str` conversion;
`int(2.83)`	`float` to `int`; discards non-integer digits;
`round(2.83)`	`float` rounded to the nearest `int`;
`round(2.83, 1)`	rounds the number to one decimal place, without changing its (`float`) type;
`complex('1+2j')`	`str` to `complex` conversion;
`float('-2.6')`	`str` to `float` conversion;
`float('1.6e8')`	scientific notation: 1.6e8 means $1.6 \cdot 10^8$;
`int('-123')`	`str` to `int` conversion;
`int('101', 2)`	binary (base 2) number as `str`, to `int`;
`bin(18)`	`int` to `str` in a binary system.

Exercises:

1. add binary numbers $101_2 + 1011_2$.
   Code: `int('101', 2) + int('1011', 2)`
2. How many decimal digits does 99^{99} have? (hint: `len(s)` returns the length of the string `s`) Code: `len(str(99**99))`

Built-in numeric types – summary

- `int` – arbitrary-precision integer
- `float` – rational number in binary floating point representation (usually according to IEEE-754 “double precision” standard)
- `complex` – complex number represented by two `float`s

Remark 1

Thanks to reasonable selection of result types by operations, developer usually does not have to care about types they use.

Remark 2

Usually we use `desired_type(something)` to convert `something` to `desired_type`.

Built-in numeric types – summary

- `int` – arbitrary-precision integer
- `float` – rational number in binary floating point representation (usually according to IEEE-754 “double precision” standard)
- `complex` – complex number represented by two `float`s

Remark 1

Thanks to reasonable selection of result types by operations, developer usually does not have to care about types they use.

Remark 2

Usually we use `desired_type(something)` to convert `something` to `desired_type`.

Built-in numeric types – summary

- `int` – arbitrary-precision integer
- `float` – rational number in binary floating point representation (usually according to IEEE-754 “double precision” standard)
- `complex` – complex number represented by two `float`s

Remark 1

Thanks to reasonable selection of result types by operations, developer usually does not have to care about types they use.

Remark 2

Usually we use `desired_type(something)` to convert `something` to `desired_type`.

Built-in numeric types – summary

- `int` – arbitrary-precision integer
- `float` – rational number in binary floating point representation (usually according to IEEE-754 “double precision” standard)
- `complex` – complex number represented by two `float`s

Remark 1

Thanks to reasonable selection of result types by operations, developer usually does not have to care about types they use.

Remark 2

Usually we use `desired_type(something)` to convert `something` to `desired_type`.

Built-in numeric types – summary

- **`int`** – arbitrary-precision integer
- **`float`** – rational number in binary floating point representation (usually according to IEEE-754 "double precision" standard)
- **`complex`** – complex number represented by two **`float`**s

Remark 1

Thanks to reasonable selection of result types by operations, developer usually does not have to care about types they use.

Remark 2

Usually we use `desired_type(something)` to convert `something` to `desired_type`.

The math module – additional mathematical functions

- The `math` module provides access to additional mathematical functions.

  Exercise: type `help('math')` to find out what functions are available.

- The functions included in `math` module cannot be used with `complex` numbers.

  Use the functions of the same name from the `cmath` module if you require support for complex numbers.

  Exercise: take a look at `help('cmath')`.

The math module – additional mathematical functions

- The `math` module provides access to additional mathematical functions.
  Exercise: type `help('math')` to find out what functions are available.
- The functions included in `math` module cannot be used with `complex` numbers.
  Use the functions of the same name from the `cmath` module if you require support for complex numbers.
  Exercise: take a look at `help('cmath')`.

The math module – additional mathematical functions

- The `math` module provides access to additional mathematical functions.
  Exercise: type `help('math')` to find out what functions are available.
- The functions included in `math` module cannot be used with `complex` numbers.
  Use the functions of the same name from the `cmath` module if you require support for complex numbers.
  Exercise: take a look at `help('cmath')`.

The math module – additional mathematical functions

- The `math` module provides access to additional mathematical functions.
  Exercise: type `help('math')` to find out what functions are available.
- The functions included in `math` module cannot be used with `complex` numbers.
  Use the functions of the same name from the `cmath` module if you require support for complex numbers.
  Exercise: take a look at `help('cmath')`.

Importing module

In order to use any module, you have to import it first, e.g.:

- `import math`
  imports the whole `math` module. After that, you can type `math.sth` to use `sth` from the module, e.g: `math.sin(0)`
- `import math as m`
  is similar, but shorter `m` prefix can be used, e.g. `m.sin(0)`
- `from math import sin, cos`
  imports **particular names** (`sin` and `cos`) **into the current namespace**, which allows for using them **without** any prefix, e.g.: `sin(0)`
- `from math import *`
  imports **all** names from the `math` module into the current namespace. Everything can be used without any prefix.
- `from math import sin as s`
  imports `sin` and makes it accessible as `s`, e.g. `s(0)`

Importing module

In order to use any module, you have to import it first, e.g.:

- `import math`
  imports the whole `math` module. After that, you can type `math.sth` to use `sth` from the module, e.g: `math.sin(0)`
- `import math as m`
  is similar, but shorter `m` prefix can be used, e.g. `m.sin(0)`
- `from math import sin, cos`
  imports **particular names** (`sin` and `cos`) **into the current namespace**, which allows for using them **without** any prefix, e.g.: `sin(0)`
- `from math import *`
  imports **all** names from the math module into the current namespace. Everything can be used without any prefix.
- `from math import sin as s`
  imports `sin` and makes it accessible as `s`, e.g. `s(0)`

Importing module

In order to use any module, you have to import it first, e.g.:

- `import math`
  imports the whole `math` module. After that, you can type `math.sth` to use `sth` from the module, e.g: `math.sin(0)`
- `import math as m`
  is similar, but shorter `m` prefix can be used, e.g. `m.sin(0)`
- `from math import sin, cos`
  imports **particular names** (`sin` and `cos`) **into the current namespace**, which allows for using them **without** any prefix, e.g.: `sin(0)`
- `from math import *`
  imports **all** names from the `math` module into the current namespace. Everything can be used without any prefix.
- `from math import sin as s`
  imports `sin` and makes it accessible as `s`, e.g. `s(0)`

Importing module

In order to use any module, you have to import it first, e.g.:

- `import math`
  imports the whole `math` module. After that, you can type `math.sth` to use `sth` from the module, e.g: `math.sin(0)`
- `import math as m`
  is similar, but shorter `m` prefix can be used, e.g. `m.sin(0)`
- `from math import sin, cos`
  imports **particular names** (`sin` and `cos`) **into the current namespace**, which allows for using them **without** any prefix, e.g.: `sin(0)`
- `from math import *`
  imports **all** names from the `math` module into the current namespace. Everything can be used without any prefix.
- `from math import sin as s`
  imports `sin` and makes it accessible as `s`, e.g. `s(0)`

Importing module

In order to use any module, you have to import it first, e.g.:

- `import math`
  imports the whole `math` module. After that, you can type `math.sth` to use `sth` from the module, e.g: `math.sin(0)`
- `import math as m`
  is similar, but shorter `m` prefix can be used, e.g. `m.sin(0)`
- `from math import sin, cos`
  imports **particular names** (`sin` and `cos`) **into the current namespace**, which allows for using them **without** any prefix, e.g.: `sin(0)`
- `from math import *`
  imports **all** names from the `math` module into the current namespace. Everything can be used without any prefix.
- `from math import sin as s`
  imports `sin` and makes it accessible as `s`, e.g. `s(0)`

Using the math module – exercises

Calculate:

1. $\cos^2(\pi/3)$
2. $\lceil 5 \cdot \log_2(20) \rceil$ (where $\lceil x \rceil$ denotes the ceiling of x)
3. $30!$
4. $e^{15.5}$
5. check if $0.1 + 0.2$ equals 0.3 (hint: due to `float` inaccuracy, you should only check if the numbers are close to each other)

Tip

After `import math` you can type `math.` and press `tab` key to see list of symbols included into the `math` module.

Using the math module – exercises

Calculate: after `import math`:

1. $\cos^2(\pi/3)$
   `math.cos(math.pi/3)**2`
2. $\lceil 5 \cdot \log_2(20) \rceil$ (where $\lceil x \rceil$ denotes the ceiling of x)
   `math.ceil(5 * math.log2(20))`
3. 30!
   `math.factorial(30)`
4. $e^{15.5}$
   `math.e ** 15.5` or (better) `math.exp(15.5)`
5. check if $0.1 + 0.2$ equals 0.3 (hint: due to **`float`** inaccuracy, you should only check if the numbers are close to each other)
   `math.isclose(0.1+0.2, 0.3)`
   Note that `0.1+0.2 == 0.3` gives `False`!

Tip

After **`import`** `math` you can type `math.` and press `tab` key to see list of symbols included into the `math` module.

Using variables – an example and an exercise

Example: The following code calculates $|2b\sin^5(a+b)+\frac{a+b}{b-a}|$, where $a=3\sqrt{2.1}$, $b=2\cos^3(\frac{\pi}{7})$:

```
import math
a = 3 * math.sqrt(2.1)
b = 2 * math.cos(math.pi/7)**3
abs(2*b*math.sin(a+b)**5 + (a+b)/(b-a))
```

The final result: `2.0715515986265305`

Exercise: calculate $ac\sin^2(ab)+\lfloor\frac{c}{a+c}\rfloor\cos^b(a)-bc$, where $a=\frac{\pi}{2}$, $b=\sin^2(\frac{\pi}{4})$, $c=e^3$.

The final result: `5.73237534872939`

Using variables – an example and an exercise

Example: The following code calculates $|2b\sin^5(a+b)+\frac{a+b}{b-a}|$, where $a=3\sqrt{2.1}$, $b=2\cos^3(\frac{\pi}{7})$:

```
import math
a = 3 * math.sqrt(2.1)
b = 2 * math.cos(math.pi/7)**3
abs(2*b*math.sin(a+b)**5 + (a+b)/(b-a))
```

The final result: `2.0715515986265305`

Exercise: calculate $ac\sin^2(ab)+\lfloor\frac{c}{a+c}\rfloor\cos^b(a)-bc$, where $a=\frac{\pi}{2}$, $b=\sin^2(\frac{\pi}{4})$, $c=e^3$.

The final result: `5.73237534872939`

Using variables – an example and an exercise

Example: The following code calculates $|2b\sin^5(a+b)+\frac{a+b}{b-a}|$, where $a=3\sqrt{2.1}$, $b=2\cos^3(\frac{\pi}{7})$:

```
import math
a = 3 * math.sqrt(2.1)
b = 2 * math.cos(math.pi/7)**3
abs(2*b*math.sin(a+b)**5 + (a+b)/(b-a))
```

The final result: `2.0715515986265305`

Exercise: calculate $ac\sin^2(ab)+\lfloor\frac{c}{a+c}\rfloor\cos^b(a)-bc$, where $a=\frac{\pi}{2}$, $b=\sin^2(\frac{\pi}{4})$, $c=e^3$.

```
import math
a = math.pi / 2
b = math.sin(math.pi / 4) ** 2
c = math.exp(3)
a*c*math.sin(a*b)**2 + c//(a+c)*math.cos(a)**b - b*c
```

The final result: `5.73237534872939`

Homework

Display help about the *statistics* module.

Calculate:

1. $2\sqrt[5]{7\sin(\pi/2)+\cos(0)/3}-\log_2(18)$;
2. the number of decimal digits of $(30!)^{11}$;
3. greatest common divisor of 60! and 8^{120};
4. the product of ternary numbers: $2021_3 \cdot 10212_3$
5. $b\tan^c(2.1a)/3e^b-\cos(a+c)$, where $a=\frac{\pi}{7}$, $b=e^2$, $c=\frac{3}{\pi}$;
6. $e^{2j}+\sqrt{-5}$, where j is the imaginary unit.
   (Hint: use the `cmath` module.)

Measure the time which Python needs to solve the last task (to calculate $e^{2j}+\sqrt{-5}$).

Please note all the expressions you used.

Bibliography

- *IPython Documentation* available on http://ipython.readthedocs.io/en/stable/
- Official *Python documentation* available on https://docs.python.org/3/, modules: *math*, *cmath*