Iterable objects in Python (3.x)

sequences, generator expressions, comprehensions, and reducing

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`

Built-in range function

`range(start, stop, step=1)` returns an iterable object that produces a sequence of integers from `start` (inclusive) to `stop` (exclusive) by `step` (which is 1 by default). For example:

- `range(3, 7)` produces $3, 4, 5, 6$;
- `range(3, 7, 2)` produces $3, 5$;
- `range(7, 2, -1)` produces $7, 6, 5, 4, 3$.

Notice that the arguments are analogous to the slice syntax.

`range(stop)` returns an iterable object that produces a sequence of integers from 0 (inclusive) to `stop` (exclusive). For example:

- `range(5)` produces $0, 1, 2, 3, 4$;
- `range(len(s))` produces all valid indices of `s`, if `s` is for example a `list` or a `tuple` instance.

Built-in range function

`range(start, stop, step=1)` returns an iterable object that produces a sequence of integers from `start` (inclusive) to `stop` (exclusive) by `step` (which is 1 by default). For example:

- `range(3, 7)` produces $3, 4, 5, 6$;
- `range(3, 7, 2)` produces $3, 5$;
- `range(7, 2, -1)` produces $7, 6, 5, 4, 3$.

Notice that the arguments are analogous to the slice syntax.

`range(stop)` returns an iterable object that produces a sequence of integers from 0 (inclusive) to `stop` (exclusive). For example:

- `range(5)` produces $0, 1, 2, 3, 4$;
- `range(len(s))` produces all valid indices of `s`, if `s` is for example a `list` or a `tuple` instance.

Built-in range function

`range(start, stop, step=1)` returns an iterable object that produces a sequence of integers from `start` (inclusive) to `stop` (exclusive) by `step` (which is 1 by default). For example:

- `range(3, 7)` produces $3, 4, 5, 6$;
- `range(3, 7, 2)` produces $3, 5$;
- `range(7, 2, -1)` produces $7, 6, 5, 4, 3$.

Notice that the arguments are analogous to the slice syntax.

`range(stop)` returns an iterable object that produces a sequence of integers from 0 (inclusive) to `stop` (exclusive). For example:

- `range(5)` produces $0, 1, 2, 3, 4$;
- `range(len(s))` produces all valid indices of `s`, if `s` is for example a `list` or a `tuple` instance.

Built-in range function

`range(start, stop, step=1)` returns an iterable object that produces a sequence of integers from `start` (inclusive) to `stop` (exclusive) by `step` (which is 1 by default). For example:

- `range(3, 7)` produces $3, 4, 5, 6$;
- `range(3, 7, 2)` produces $3, 5$;
- `range(7, 2, -1)` produces $7, 6, 5, 4, 3$.

Notice that the arguments are analogous to the slice syntax.

`range(stop)` returns an iterable object that produces a sequence of integers from 0 (inclusive) to `stop` (exclusive). For example:

- `range(5)` produces $0, 1, 2, 3, 4$;
- `range(len(s))` produces all valid indices of `s`, if `s` is for example a `list` or a `tuple` instance.

Built-in range function

`range(start, stop, step=1)` returns an iterable object that produces a sequence of integers from `start` (inclusive) to `stop` (exclusive) by `step` (which is 1 by default). For example:

- `range(3, 7)` produces $3, 4, 5, 6$;
- `range(3, 7, 2)` produces $3, 5$;
- `range(7, 2, -1)` produces $7, 6, 5, 4, 3$.

Notice that the arguments are analogous to the slice syntax.

`range(stop)` returns an iterable object that produces a sequence of integers from 0 (inclusive) to `stop` (exclusive). For example:

- `range(5)` produces $0, 1, 2, 3, 4$;
- `range(len(s))` produces all valid indices of `s`, if `s` is for example a `list` or a `tuple` instance.

Iterable objects

- `list`, `tuple` and the result of `range` are examples of iterable objects – they can deliver a sequence of values one at a time;
- `range` is usually more memory efficient than `list` and `tuple`, because `range` result does not store all values it delivers, but instead it produces them on demand, one by one;
- functions which process sequences of values, usually accept iterable objects of any types as the arguments;
- examples of such functions are `sum`, `max`, `min`, and the functions included in the `statistics` module, like: `mean`, `median`, `mode`, `stdev`, or `variance`.
- iterable objects can be used to construct `list`s or `tuple`s;
- `list(iterable)` or `tuple(iterable)` construct, respectively, `list` or `tuple` initialized from `iterable`'s items,
- e.g. `list(range(5, 10))` gives `[5, 6, 7, 8, 9]`.

Iterable objects

- `list`, `tuple` and the result of `range` are examples of iterable objects – they can deliver a sequence of values one at a time;
- `range` is usually more memory efficient than `list` and `tuple`, because `range` result does not store all values it delivers, but instead it produces them on demand, one by one;
- functions which process sequences of values, usually accept iterable objects of any types as the arguments;
- examples of such functions are `sum`, `max`, `min`, and the functions included in the `statistics` module, like: `mean`, `median`, `mode`, `stdev`, or `variance`.
- iterable objects can be used to construct `lists` or `tuples`;
- `list(iterable)` or `tuple(iterable)` construct, respectively, `list` or `tuple` initialized from `iterable`'s items,
- e.g. `list(range(5, 10))` gives `[5, 6, 7, 8, 9]`.

Iterable objects

- `list`, `tuple` and the result of `range` are examples of iterable objects – they can deliver a sequence of values one at a time;
- `range` is usually more memory efficient than `list` and `tuple`, because `range` result does not store all values it delivers, but instead it produces them on demand, one by one;
- functions which process sequences of values, usually accept iterable objects of any types as the arguments;
- examples of such functions are `sum`, `max`, `min`, and the functions included in the `statistics` module, like: `mean`, `median`, `mode`, `stdev`, or `variance`.
- iterable objects can be used to construct `lists` or `tuples`;
- `list(iterable)` or `tuple(iterable)` construct, respectively, `list` or `tuple` initialized from `iterable`'s items,
- e.g. `list(range(5, 10))` gives `[5, 6, 7, 8, 9]`.

Iterable objects

- `list`, `tuple` and the result of `range` are examples of iterable objects – they can deliver a sequence of values one at a time;
- `range` is usually more memory efficient than `list` and `tuple`, because `range` result does not store all values it delivers, but instead it produces them on demand, one by one;
- functions which process sequences of values, usually accept iterable objects of any types as the arguments;
- examples of such functions are `sum`, `max`, `min`, and the functions included in the `statistics` module, like: `mean`, `median`, `mode`, `stdev`, or `variance`.
- iterable objects can be used to construct `lists` or `tuples`;
- `list(iterable)` or `tuple(iterable)` construct, respectively, `list` or `tuple` initialized from `iterable`'s items,
- e.g. `list(range(5, 10))` gives `[5, 6, 7, 8, 9]`.

Iterable objects

- **`list`**, **`tuple`** and the result of **`range`** are examples of iterable objects – they can deliver a sequence of values one at a time;
- **`range`** is usually more memory efficient than **`list`** and **`tuple`**, because **`range`** result does not store all values it delivers, but instead it produces them on demand, one by one;
- functions which process sequences of values, usually accept iterable objects of any types as the arguments;
- examples of such functions are **`sum`**, **`max`**, **`min`**, and the functions included in the `statistics` module, like: `mean`, `median`, `mode`, `stdev`, or `variance`.
- iterable objects can be used to construct **`list`**s or **`tuple`**s;
- **`list`**`(iterable)` or **`tuple`**`(iterable)` construct, respectively, **`list`** or **`tuple`** initialized from `iterable`'s items,
- e.g. **`list`**`(`**`range`**`(5, 10))` gives `[5, 6, 7, 8, 9]`.

Iterable objects

- **`list`**, **`tuple`** and the result of **`range`** are examples of iterable objects – they can deliver a sequence of values one at a time;
- **`range`** is usually more memory efficient than **`list`** and **`tuple`**, because **`range`** result does not store all values it delivers, but instead it produces them on demand, one by one;
- functions which process sequences of values, usually accept iterable objects of any types as the arguments;
- examples of such functions are **`sum`**, **`max`**, **`min`**, and the functions included in the `statistics` module, like: `mean`, `median`, `mode`, `stdev`, or `variance`.
- iterable objects can be used to construct **`list`**s or **`tuple`**s;
- **`list`**`(iterable)` or **`tuple`**`(iterable)` construct, respectively, **`list`** or **`tuple`** initialized from `iterable`'s items,
- e.g. **`list`**`(`**`range`**`(5, 10))` gives `[5, 6, 7, 8, 9]`.

Iterable objects

- **`list`**, **`tuple`** and the result of **`range`** are examples of iterable objects – they can deliver a sequence of values one at a time;
- **`range`** is usually more memory efficient than **`list`** and **`tuple`**, because **`range`** result does not store all values it delivers, but instead it produces them on demand, one by one;
- functions which process sequences of values, usually accept iterable objects of any types as the arguments;
- examples of such functions are **`sum`**, **`max`**, **`min`**, and the functions included in the `statistics` module, like: `mean`, `median`, `mode`, `stdev`, or `variance`.
- iterable objects can be used to construct **`list`**s or **`tuple`**s;
- **`list`**`(iterable)` or **`tuple`**`(iterable)` construct, respectively, **`list`** or **`tuple`** initialized from `iterable`'s items,
- e.g. **`list`**`(`**`range`**`(5, 10))` gives `[5, 6, 7, 8, 9]`.

sum, min, max – examples and exercises

Execute:

`sum(range(3,7))`	sums items produced by `range`; $\sum_{i=3}^{6} i$;
`sum([1,2,3])`	sums `list` elements; $1+2+3$;
`sum((1,2,3))`	sums `tuple` elements; $1+2+3$;
`sum(1,2,3)`	throws `TypeError` since `sum` cannot accept 3 separate arguments;
`sum((1,"ab",2))`	also `TypeError`; cannot add `int` to `str`.
`min(range(1,8,2))`	the smallest item produced by `range`;
`max((1, 2, 3))`	the largest elements of `tuple`;
`max(1,2,3)`	`max` and `min` accept separate arguments;
`min((1,"ab",2))`	`TypeError`; cannot compare `int` to `str`.

Calculate:

1. $\sum_{i=0}^{100} i = 0 + 1 + \cdots + 100$;
2. $10 + 15 + 20 + \cdots + 100$;
3. $\max(2^{20}, 15!)$ (hint: `import math`)

sum, min, max – examples and exercises

Execute:

`sum(range(3,7))`	sums items produced by `range`; $\sum_{i=3}^{6} i$;
`sum([1,2,3])`	sums `list` elements; $1+2+3$;
`sum((1,2,3))`	sums `tuple` elements; $1+2+3$;
`sum(1,2,3)`	throws `TypeError` since `sum` cannot accept 3 separate arguments;
`sum((1,"ab",2))`	also `TypeError`; cannot add `int` to `str`.
`min(range(1,8,2))`	the smallest item produced by `range`;
`max((1, 2, 3))`	the largest elements of `tuple`;
`max(1,2,3)`	`max` and `min` accept separate arguments;
`min((1,"ab",2))`	`TypeError`; cannot compare `int` to `str`.

Calculate:

1. $\sum_{i=0}^{100} i = 0 + 1 + \cdots + 100$;
2. $10 + 15 + 20 + \cdots + 100$;
3. $\max(2^{20}, 15!)$ (hint: `import math`)

sum, min, max – examples and exercises

Execute:

`sum(range(3,7))`	sums items produced by `range`; $\sum_{i=3}^{6} i$;
`sum([1,2,3])`	sums `list` elements; $1+2+3$;
`sum((1,2,3))`	sums `tuple` elements; $1+2+3$;
`sum(1,2,3)`	throws `TypeError` since `sum` cannot accept 3 separate arguments;
`sum((1,"ab",2))`	also `TypeError`; cannot add `int` to `str`.
`min(range(1,8,2))`	the smallest item produced by `range`;
`max((1, 2, 3))`	the largest elements of `tuple`;
`max(1,2,3)`	`max` and `min` accept separate arguments;
`min((1,"ab",2))`	`TypeError`; cannot compare `int` to `str`.

Calculate:

1. $\sum_{i=0}^{100} i = 0 + 1 + \cdots + 100$;
2. $10 + 15 + 20 + \cdots + 100$;
3. $\max(2^{20}, 15!)$ (hint: `import math`)

sum, min, max – examples and exercises

Execute:

`sum(range(3,7))`	sums items produced by `range`; $\sum_{i=3}^{6} i$;
`sum([1,2,3])`	sums `list` elements; $1+2+3$;
`sum((1,2,3))`	sums `tuple` elements; $1+2+3$;
`sum(1,2,3)`	throws `TypeError` since `sum` cannot accept 3 separate arguments;
`sum((1,"ab",2))`	also `TypeError`; cannot add `int` to `str`.
`min(range(1,8,2))`	the smallest item produced by `range`;
`max((1, 2, 3))`	the largest elements of `tuple`;
`max(1,2,3)`	`max` and `min` accept separate arguments;
`min((1,"ab",2))`	`TypeError`; cannot compare `int` to `str`.

Calculate:

1. $\sum_{i=0}^{100} i = 0 + 1 + \cdots + 100$;
2. $10 + 15 + 20 + \cdots + 100$;
3. $\max(2^{20}, 15!)$ (hint: `import math`)

sum, min, max – examples and exercises

Execute:

`sum(range(3,7))`	sums items produced by `range`; $\sum_{i=3}^{6} i$;
`sum([1,2,3])`	sums `list` elements; $1+2+3$;
`sum((1,2,3))`	sums `tuple` elements; $1+2+3$;
`sum(1,2,3)`	throws `TypeError` since `sum` cannot accept 3 separate arguments;
`sum((1,"ab",2))`	also `TypeError`; cannot add `int` to `str`.
`min(range(1,8,2))`	the smallest item produced by `range`;
`max((1, 2, 3))`	the largest elements of `tuple`;
`max(1,2,3)`	`max` and `min` accept separate arguments;
`min((1,"ab",2))`	`TypeError`; cannot compare `int` to `str`.

Calculate:

1. $\sum_{i=0}^{100} i = 0 + 1 + \cdots + 100$;
2. $10 + 15 + 20 + \cdots + 100$;
3. $\max(2^{20}, 15!)$ (hint: `import math`)

sum, min, max – examples and exercises

Execute:

`sum(range(3,7))`	sums items produced by `range`; $\sum_{i=3}^{6} i$;
`sum([1,2,3])`	sums `list` elements; $1+2+3$;
`sum((1,2,3))`	sums `tuple` elements; $1+2+3$;
`sum(1,2,3)`	throws `TypeError` since `sum` cannot accept 3 separate arguments;
`sum((1,"ab",2))`	also `TypeError`; cannot add `int` to `str`.
`min(range(1,8,2))`	the smallest item produced by `range`;
`max((1, 2, 3))`	the largest elements of `tuple`;
`max(1,2,3)`	`max` and `min` accept separate arguments;
`min((1,"ab",2))`	`TypeError`; cannot compare `int` to `str`.

Calculate:

1. $\sum_{i=0}^{100} i = 0 + 1 + \cdots + 100$;
2. $10 + 15 + 20 + \cdots + 100$;
3. $\max(2^{20}, 15!)$ (hint: `import math`)

sum, min, max – examples and exercises

Execute:

`sum(range(3,7))`	sums items produced by **`range`**; $\sum_{i=3}^{6} i$;
`sum([1,2,3])`	sums **`list`** elements; $1+2+3$;
`sum((1,2,3))`	sums **`tuple`** elements; $1+2+3$;
`sum(1,2,3)`	throws `TypeError` since **`sum`** cannot accept 3 separate arguments;
`sum((1,"ab",2))`	also `TypeError`; cannot add **`int`** to **`str`**.
`min(range(1,8,2))`	the smallest item produced by **`range`**;
`max((1, 2, 3))`	the largest elements of **`tuple`**;
`max(1,2,3)`	**`max`** and **`min`** accept separate arguments;
`min((1,"ab",2))`	`TypeError`; cannot compare **`int`** to **`str`**.

Calculate:

1. $\sum_{i=0}^{100} i = 0 + 1 + \cdots + 100$;
2. $10 + 15 + 20 + \cdots + 100$;
3. $\max(2^{20}, 15!)$ (hint: `import math`)

sum, min, max – examples and exercises

Execute:

`sum(range(3,7))`	sums items produced by `range`; $\sum_{i=3}^{6} i$;
`sum([1,2,3])`	sums `list` elements; $1+2+3$;
`sum((1,2,3))`	sums `tuple` elements; $1+2+3$;
`sum(1,2,3)`	throws `TypeError` since `sum` cannot accept 3 separate arguments;
`sum((1,"ab",2))`	also `TypeError`; cannot add `int` to `str`.
`min(range(1,8,2))`	the smallest item produced by `range`;
`max((1, 2, 3))`	the largest elements of `tuple`;
`max(1,2,3)`	`max` and `min` accept separate arguments;
`min((1,"ab",2))`	`TypeError`; cannot compare `int` to `str`.

Calculate:

1. $\sum_{i=0}^{100} i = 0 + 1 + \cdots + 100$;
2. $10 + 15 + 20 + \cdots + 100$;
3. $\max(2^{20}, 15!)$ (hint: `import math`)

sum, min, max – examples and exercises

Execute:

`sum(range(3,7))`	sums items produced by `range`; $\sum_{i=3}^{6} i$;
`sum([1,2,3])`	sums `list` elements; $1+2+3$;
`sum((1,2,3))`	sums `tuple` elements; $1+2+3$;
`sum(1,2,3)`	throws `TypeError` since `sum` cannot accept 3 separate arguments;
`sum((1,"ab",2))`	also `TypeError`; cannot add `int` to `str`.
`min(range(1,8,2))`	the smallest item produced by `range`;
`max((1, 2, 3))`	the largest elements of `tuple`;
`max(1,2,3)`	`max` and `min` accept separate arguments;
`min((1,"ab",2))`	`TypeError`; cannot compare `int` to `str`.

Calculate:

1. $\sum_{i=0}^{100} i = 0 + 1 + \cdots + 100$;
2. $10 + 15 + 20 + \cdots + 100$;
3. $\max(2^{20}, 15!)$ (hint: `import math`)

sum, min, max – examples and exercises

Execute:

`sum(range(3,7))`	sums items produced by **`range`**; $\sum_{i=3}^{6} i$;
`sum([1,2,3])`	sums **`list`** elements; $1+2+3$;
`sum((1,2,3))`	sums **`tuple`** elements; $1+2+3$;
`sum(1,2,3)`	throws `TypeError` since **`sum`** cannot accept 3 separate arguments;
`sum((1,"ab",2))`	also `TypeError`; cannot add **`int`** to **`str`**.
`min(range(1,8,2))`	the smallest item produced by **`range`**;
`max((1, 2, 3))`	the largest elements of **`tuple`**;
`max(1,2,3)`	**`max`** and **`min`** accept separate arguments;
`min((1,"ab",2))`	`TypeError`; cannot compare **`int`** to **`str`**.

Calculate:

1. $\sum_{i=0}^{100} i = 0 + 1 + \cdots + 100$;
2. $10 + 15 + 20 + \cdots + 100$;
3. $\max(2^{20}, 15!)$ (hint: **`import`** `math`)

sum, min, max – examples and exercises

Execute:

`sum(range(3,7))`	sums items produced by **`range`**; $\sum_{i=3}^{6} i$;
`sum([1,2,3])`	sums **`list`** elements; $1+2+3$;
`sum((1,2,3))`	sums **`tuple`** elements; $1+2+3$;
`sum(1,2,3)`	throws `TypeError` since **`sum`** cannot accept 3 separate arguments;
`sum((1,"ab",2))`	also `TypeError`; cannot add **`int`** to **`str`**.
`min(range(1,8,2))`	the smallest item produced by **`range`**;
`max((1, 2, 3))`	the largest elements of **`tuple`**;
`max(1,2,3)`	**`max`** and **`min`** accept separate arguments;
`min((1,"ab",2))`	`TypeError`; cannot compare **`int`** to **`str`**.

Calculate:

1. $\sum_{i=0}^{100} i = 0+1+\cdots+100$; `sum(range(0, 101))`
2. $10+15+20+\cdots+100$; `sum(range(10, 101, 5))`
3. $\max(2^{20}, 15!)$ (hint: `import math`)
   `max(2**20, math.factorial(15))`

key argument for min and max

- `min` and `max` can take an optional `key` parameter;
- `key` supplies a single-argument function which is used to extract a comparison key from each item;
- if `key` is given, `min`/`max` returns the item with the smallest/largest comparison key.

For example:

- `max((1,-3,2), key=abs)` or `max(1,-3,2, key=abs)` return item with the largest absolute value (`-3`);
- `min(['a', 'cd', 'x'], key=len)` returns the (first) shortest string (`'a'`) in the list;
- `min(5,6,7,8, key=lambda x: x % 3)` returns number with the smallest remainder after division by 3;
  `lambda x: x % 3` defines a single-argument function which returns remainder after division of its argument (`x`) by 3;

key argument for min and max

- `min` and `max` can take an optional `key` parameter;
- `key` supplies a single-argument function which is used to extract a comparison key from each item;
- if `key` is given, `min`/`max` returns the item with the smallest/largest comparison key.

For example:

- `max((1,-3,2), key=abs)` or `max(1,-3,2, key=abs)` return item with the largest absolute value (`-3`);
- `min(['a', 'cd', 'x'], key=len)` returns the (first) shortest string (`'a'`) in the list;
- `min(5,6,7,8, key=lambda x: x % 3)` returns number with the smallest remainder after division by 3;
  `lambda x: x % 3` defines a single-argument function which returns remainder after division of its argument (`x`) by 3;

key argument for min and max

- **`min`** and **`max`** can take an optional `key` parameter;
- `key` supplies a single-argument function which is used to extract a comparison key from each item;
- if `key` is given, **`min`**/**`max`** returns the item with the smallest/largest comparison key.

For example:

- **`max`**`((1,-3,2), key=`**`abs`**`)` or **`max`**`(1,-3,2, key=`**`abs`**`)` return item with the largest absolute value (`-3`);
- **`min`**`(['a', 'cd', 'x'], key=`**`len`**`)` returns the (first) shortest string (`'a'`) in the list;
- **`min`**`(5,6,7,8, key=`**`lambda`**` x: x % 3)` returns number with the smallest remainder after division by 3;
  `lambda`` x: x % 3` defines a single-argument function which returns remainder after division of its argument (`x`) by 3;

key argument for min and max

- `min` and `max` can take an optional `key` parameter;
- `key` supplies a single-argument function which is used to extract a comparison key from each item;
- if `key` is given, `min`/`max` returns the item with the smallest/largest comparison key.

For example:

- `max((1,-3,2), key=abs)` or `max(1,-3,2, key=abs)` return item with the largest absolute value (`-3`);
- `min(['a', 'cd', 'x'], key=len)` returns the (first) shortest string (`'a'`) in the list;
- `min(5,6,7,8, key=lambda x: x % 3)` returns number with the smallest remainder after division by 3;
  `lambda x: x % 3` defines a single-argument function which returns remainder after division of its argument (`x`) by 3;

key argument for min and max

- `min` and `max` can take an optional `key` parameter;
- `key` supplies a single-argument function which is used to extract a comparison key from each item;
- if `key` is given, `min`/`max` returns the item with the smallest/largest comparison key.

For example:

- `max((1,-3,2), key=abs)` or `max(1,-3,2, key=abs)` return item with the largest absolute value (`-3`);
- `min(['a', 'cd', 'x'], key=len)` returns the (first) shortest string (`'a'`) in the list;
- `min(5,6,7,8, key=lambda x: x % 3)` returns number with the smallest remainder after division by 3;
  `lambda x: x % 3` defines a single-argument function which returns remainder after division of its argument (`x`) by 3;

key argument for min and max

- `min` and `max` can take an optional `key` parameter;
- `key` supplies a single-argument function which is used to extract a comparison key from each item;
- if `key` is given, `min`/`max` returns the item with the smallest/largest comparison key.

For example:

- `max((1,-3,2), key=abs)` or `max(1,-3,2, key=abs)` return item with the largest absolute value (`-3`);
- `min(['a', 'cd', 'x'], key=len)` returns the (first) shortest string (`'a'`) in the list;
- `min(5,6,7,8, key=lambda x: x % 3)` returns number with the smallest remainder after division by 3;
  `lambda x: x % 3` defines a single-argument function which returns remainder after division of its argument (`x`) by 3;

key argument for min and max

- `min` and `max` can take an optional `key` parameter;
- `key` supplies a single-argument function which is used to extract a comparison key from each item;
- if `key` is given, `min`/`max` returns the item with the smallest/largest comparison key.

For example:

- `max((1,-3,2), key=abs)` or `max(1,-3,2, key=abs)` return item with the largest absolute value (`-3`);
- `min(['a', 'cd', 'x'], key=len)` returns the (first) shortest string (`'a'`) in the list;
- `min(5,6,7,8, key=lambda x: x % 3)` returns number with the smallest remainder after division by 3;
  `lambda x: x % 3` defines a single-argument function which returns remainder after division of its argument (`x`) by 3;

key argument for min and max – an example and exercises

Let us construct the **`list`**:

```
a=[[1,6], [8], (3,4,4)]
```

Example: `max(a, key=len)` returns the longest member of `a`.

Exercise: find in `a`:

1. the shortest member
2. the member with the smallest sum of elements
3. the member with the smallest maximum element
4. the member with the largest first element
5. the member with the smallest last element

key argument for min and max – an example and exercises

Let us construct the **`list`**:

```
a=[[1,6], [8], (3,4,4)]
```

Example: `max(a, key=len)` returns the longest member of `a`.

Exercise: find in `a`:

1. the shortest member
2. the member with the smallest sum of elements
3. the member with the smallest maximum element
4. the member with the largest first element
5. the member with the smallest last element

key argument for min and max – an example and exercises

Let us construct the **`list`**:

```
a=[[1,6], [8], (3,4,4)]
```

Example: `max(a, key=len)` returns the longest member of `a`.

Exercise: find in `a`:

1. the shortest member
2. the member with the smallest sum of elements
3. the member with the smallest maximum element
4. the member with the largest first element
5. the member with the smallest last element

key argument for min and max – an example and exercises

Let us construct the **list**:

```
a=[[1,6], [8], (3,4,4)]
```

Example: `max(a, key=len)` returns the longest member of `a`.

Exercise: find in `a`:

1. the shortest member
   `min(a, key=len)`
2. the member with the smallest sum of elements
   `min(a, key=sum)`
3. the member with the smallest maximum element
   `min(a, key=max)`
4. the member with the largest first element
   `max(a, key=lambda m: m[0])`
5. the member with the smallest last element
   `min(a, key=lambda m: m[-1])`

The (universal) reduce function [1/2]

`functools.reduce(function, iterable[, initializer])`
Apply `function` of two arguments cumulatively to the items of sequence `iterable`, from left to right, so as to reduce the sequence to a single value.
If the optional `initializer` is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

Example:

- `from functools import reduce`
  since `reduce` is included in the `functools` module, we have to import it first;
- `reduce(lambda x,y: x+y, [1,2,3,4])`
  $= ((1+2)+3)+4$; equivalent of `sum([1,2,3,4])`;
  note that the first parameter (`lambda x,y: x+y`) is the lambda function which sums its two arguments;

The (universal) reduce function [1/2]

`functools.reduce(function, iterable[, initializer])`
Apply `function` of two arguments cumulatively to the items of sequence `iterable`, from left to right, so as to reduce the sequence to a single value.
If the optional `initializer` is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

Example:

- `from functools import reduce`
  since `reduce` is included in the `functools` module, we have to import it first;
- `reduce(lambda x,y: x+y, [1,2,3,4])`
  $= ((1 + 2) + 3) + 4$; equivalent of `sum([1,2,3,4])`;
  note that the first parameter (`lambda x,y: x+y`) is the lambda function which sums its two arguments;

The (universal) reduce function [1/2]

`functools.reduce(function, iterable[, initializer])`
Apply `function` of two arguments cumulatively to the items of sequence `iterable`, from left to right, so as to reduce the sequence to a single value.
If the optional `initializer` is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

Example:

- `from functools import reduce`
  since `reduce` is included in the `functools` module, we have to import it first;
- `reduce(lambda x,y: x+y, [1,2,3,4])`
  $= ((1 + 2) + 3) + 4$; equivalent of `sum([1,2,3,4])`;
  note that the first parameter (`lambda x,y: x+y`) is the lambda function which sums its two arguments;

The (universal) reduce function [2/2]

- `reduce(lambda x,y: x*y, range(5, 31))`
  $= \prod_{i=5}^{30} i = 5 \cdot 6 \cdot \ldots \cdot 30$;
- `reduce(lambda x,y: x*y, [])`
  throws `TypeError` – empty sequence with no initial value;
- `reduce(lambda x,y: x*y, [], 1)`
  returns initial value (1); a product of an empty sequence.
- `reduce(lambda x,y: x if x>y else y, (1,7,3))` and `reduce(max, (1,3,7))` are equivalents of `max((1,7,3))`;

  `x if x>y else y`
  equals `x` if `x>y`, and `y` otherwise; equals `max(x,y)`.

Exercises: Use `reduce` to calculate:

1. $\prod_{i=10}^{90} i$
2. $\min(3, 9, -2)$

The (universal) reduce function [2/2]

- `reduce(lambda x,y: x*y, range(5, 31))`
  $= \prod_{i=5}^{30} i = 5 \cdot 6 \cdot \ldots \cdot 30$;
- `reduce(lambda x,y: x*y, [])`
  throws `TypeError` – empty sequence with no initial value;
- `reduce(lambda x,y: x*y, [], 1)`
  returns initial value (1); a product of an empty sequence.
- `reduce(lambda x,y: x if x>y else y, (1,7,3))` and `reduce(max, (1,3,7))` are equivalents of `max((1,7,3))`;

  `x if x>y else y`
  equals `x` if `x>y`, and `y` otherwise; equals `max(x,y)`.

Exercises: Use `reduce` to calculate:

1. $\prod_{i=10}^{90} i$
2. $\min(3, 9, -2)$

The (universal) reduce function [2/2]

- `reduce(lambda x,y: x*y, range(5, 31))`
  $= \prod_{i=5}^{30} i = 5 \cdot 6 \cdot \ldots \cdot 30$;
- `reduce(lambda x,y: x*y, [])`
  throws `TypeError` – empty sequence with no initial value;
- `reduce(lambda x,y: x*y, [], 1)`
  returns initial value (1); a product of an empty sequence.
- `reduce(lambda x,y: x if x>y else y, (1,7,3))` and `reduce(max, (1,3,7))` are equivalents of `max((1,7,3))`;
  `x if x>y else y`
  equals `x` if `x>y`, and `y` otherwise; equals `max(x,y)`.

Exercises: Use `reduce` to calculate:

1. $\prod_{i=10}^{90} i$
2. $\min(3, 9, -2)$

The (universal) reduce function [2/2]

- `reduce(lambda x,y: x*y, range(5, 31))`
  $= \prod_{i=5}^{30} i = 5 \cdot 6 \cdot \ldots \cdot 30$;
- `reduce(lambda x,y: x*y, [])`
  throws `TypeError` – empty sequence with no initial value;
- `reduce(lambda x,y: x*y, [], 1)`
  returns initial value (1); a product of an empty sequence.
- `reduce(lambda x,y: x if x>y else y, (1,7,3))` and `reduce(max, (1,3,7))` are equivalents of `max((1,7,3))`;
  `x if x>y else y`
  equals `x` if `x>y`, and `y` otherwise; equals `max(x,y)`.

Exercises: Use `reduce` to calculate:

1. $\prod_{i=10}^{90} i$
2. $\min(3, 9, -2)$

The (universal) reduce function [2/2]

- `reduce(lambda x,y: x*y, range(5, 31))` $= \prod_{i=5}^{30} i = 5 \cdot 6 \cdot \ldots \cdot 30$;
- `reduce(lambda x,y: x*y, [])` throws `TypeError` – empty sequence with no initial value;
- `reduce(lambda x,y: x*y, [], 1)` returns initial value (1); a product of an empty sequence.
- `reduce(lambda x,y: x if x>y else y, (1,7,3))` and `reduce(max, (1,3,7))` are equivalents of `max((1,7,3))`;

  `x if x>y else y` equals `x` if `x>y`, and `y` otherwise; equals `max(x,y)`.

Exercises: Use `reduce` to calculate:

1. $\prod_{i=10}^{90} i$
2. $\min(3, 9, -2)$

The (universal) reduce function [2/2]

- `reduce(lambda x,y: x*y, range(5, 31))`
  $= \prod_{i=5}^{30} i = 5 \cdot 6 \cdot \ldots \cdot 30$;
- `reduce(lambda x,y: x*y, [])`
  throws `TypeError` – empty sequence with no initial value;
- `reduce(lambda x,y: x*y, [], 1)`
  returns initial value (1); a product of an empty sequence.
- `reduce(lambda x,y: x if x>y else y, (1,7,3))` and `reduce(max, (1,3,7))` are equivalents of `max((1,7,3))`;

  `x if x>y else y`
  equals `x` if `x>y`, and `y` otherwise; equals `max(x,y)`.

Exercises: Use `reduce` to calculate:

1. $\prod_{i=10}^{90} i$
2. $\min(3, 9, -2)$

The (universal) reduce function [2/2]

- `reduce(lambda x,y: x*y, range(5, 31))`
  $= \prod_{i=5}^{30} i = 5 \cdot 6 \cdot \ldots \cdot 30$;
- `reduce(lambda x,y: x*y, [])`
  throws `TypeError` – empty sequence with no initial value;
- `reduce(lambda x,y: x*y, [], 1)`
  returns initial value (1); a product of an empty sequence.
- `reduce(lambda x,y: x if x>y else y, (1,7,3))` and `reduce(max, (1,3,7))` are equivalents of `max((1,7,3))`;
  `x if x>y else y`
  equals `x` if `x>y`, and `y` otherwise; equals `max(x,y)`.

Exercises: Use `reduce` to calculate:

1. $\prod_{i=10}^{90} i$
2. $\min(3, 9, -2)$

The (universal) reduce function [2/2]

- `reduce(lambda x,y: x*y, range(5, 31))` $= \prod_{i=5}^{30} i = 5 \cdot 6 \cdot \ldots \cdot 30$;
- `reduce(lambda x,y: x*y, [])` throws `TypeError` – empty sequence with no initial value;
- `reduce(lambda x,y: x*y, [], 1)` returns initial value (1); a product of an empty sequence.
- `reduce(lambda x,y: x if x>y else y, (1,7,3))` and `reduce(max, (1,3,7))` are equivalents of `max((1,7,3))`;

  `x if x>y else y` equals `x` if `x>y`, and `y` otherwise; equals `max(x,y)`.

Exercises: Use `reduce` to calculate:

1. $\prod_{i=10}^{90} i$ `reduce(lambda x,y: x*y, range(10, 91))`
2. $\min(3, 9, -2)$ `reduce(min, (3,9,-2))` or `reduce(lambda x,y: x if x<y else y, (3,9,-2))`

math.fsum – accurate float sum

`math.fsum(iterable)`

Return an accurate **`float`** sum of values in the `iterable`.

Example: let us calculate $10^{100} + 0.1 - 10^{100}$ (execute:):

- `1e100 + 0.1 - 1e100` gives `0`;
- `sum((1e100, 0.1, -1e100))` gives `0`;
- $10^{100} + 0.1 = \underbrace{100\ldots0.1}_{\text{102 digits}}$ has 102 significant decimal digits,
- but `float` stores only about 16 (the most significant) of them,
- so `1e100 + 0.1 == 1e100` (check!);
- `from math import fsum`
  `fsum((1e100, 0.1, -1e100))` gives `0.1`;
- (`1e100 - 1e100 + 0.1` also gives `0.1`).

Exercise: Compare results given by:
`sum([0.1]*10)` and `fsum([0.1]*10)`

math.fsum – accurate float sum

`math.fsum(iterable)`

Return an accurate **`float`** sum of values in the `iterable`.

Example: let us calculate $10^{100} + 0.1 - 10^{100}$ (execute:):

- `1e100 + 0.1 - 1e100` gives `0`;
- **`sum`**`((1e100, 0.1, -1e100))` gives `0`;
- $10^{100} + 0.1 = \underbrace{100\ldots0.1}_{\text{102 digits}}$ has 102 significant decimal digits,
- but **`float`** stores only about 16 (the most significant) of them,
- so `1e100 + 0.1 == 1e100` (check!);
- **`from`** `math` **`import`** `fsum`
  `fsum((1e100, 0.1, -1e100))` gives `0.1`;
- (`1e100 - 1e100 + 0.1` also gives `0.1`).

Exercise: Compare results given by:
`sum``([0.1]*10)` and `fsum([0.1]*10)`

math.fsum – accurate float sum

`math.fsum(iterable)`

Return an accurate **`float`** sum of values in the `iterable`.

Example: let us calculate $10^{100} + 0.1 - 10^{100}$ (execute:):

- `1e100 + 0.1 - 1e100` gives `0`;
- `sum((1e100, 0.1, -1e100))` gives `0`;
- $10^{100} + 0.1 = \underbrace{100\ldots0.1}_{\text{102 digits}}$ has 102 significant decimal digits,
- but **`float`** stores only about 16 (the most significant) of them,
- so `1e100 + 0.1 == 1e100` (check!);
- `from math import fsum`
  `fsum((1e100, 0.1, -1e100))` gives `0.1`;
- (`1e100 - 1e100 + 0.1` also gives `0.1`).

Exercise: Compare results given by:
`sum([0.1]*10)` and `fsum([0.1]*10)`

math.fsum – accurate float sum

`math.fsum(iterable)`

Return an accurate **`float`** sum of values in the `iterable`.

Example: let us calculate $10^{100} + 0.1 - 10^{100}$ (execute:):

- `1e100 + 0.1 - 1e100` gives `0`;
- **`sum`**`((1e100, 0.1, -1e100))` gives `0`;
- $10^{100} + 0.1 = \underbrace{100\ldots0.1}_{\text{102 digits}}$ has 102 significant decimal digits,
- but **`float`** stores only about 16 (the most significant) of them,
- so `1e100 + 0.1 == 1e100` (check!);
- **`from`** `math` **`import`** `fsum`
  `fsum((1e100, 0.1, -1e100))` gives `0.1`;
- (`1e100 - 1e100 + 0.1` also gives `0.1`).

Exercise: Compare results given by:
`sum``([0.1]*10)` and `fsum([0.1]*10)`

math.fsum – accurate float sum

`math.fsum(iterable)`

Return an accurate **`float`** sum of values in the `iterable`.

Example: let us calculate $10^{100} + 0.1 - 10^{100}$ (execute:):

- `1e100 + 0.1 - 1e100` gives `0`;
- **`sum`**`((1e100, 0.1, -1e100))` gives `0`;
- $10^{100} + 0.1 = \underbrace{100\ldots0.1}_{\text{102 digits}}$ has 102 significant decimal digits,
- but **`float`** stores only about 16 (the most significant) of them,
- so `1e100 + 0.1 == 1e100` (check!);
- **`from`** `math` **`import`** `fsum`
  `fsum((1e100, 0.1, -1e100))` gives `0.1`;
- (`1e100 - 1e100 + 0.1` also gives `0.1`).

Exercise: Compare results given by:
`sum``([0.1]*10)` and `fsum([0.1]*10)`

math.fsum – accurate float sum

`math.fsum(iterable)`

Return an accurate **`float`** sum of values in the `iterable`.

Example: let us calculate $10^{100} + 0.1 - 10^{100}$ (execute:):

- `1e100 + 0.1 - 1e100` gives `0`;
- **`sum`**`((1e100, 0.1, -1e100))` gives `0`;
- $10^{100} + 0.1 = \underbrace{100\ldots0.1}_{\text{102 digits}}$ has 102 significant decimal digits,
- but **`float`** stores only about 16 (the most significant) of them,
- so `1e100 + 0.1 == 1e100` (check!);
- **`from`** `math` **`import`** `fsum`
  `fsum((1e100, 0.1, -1e100))` gives `0.1`;
- (`1e100 - 1e100 + 0.1` also gives `0.1`).

Exercise: Compare results given by:
`sum``([0.1]*10)` and `fsum([0.1]*10)`

math.fsum – accurate float sum

`math.fsum(iterable)`

Return an accurate **`float`** sum of values in the `iterable`.

Example: let us calculate $10^{100} + 0.1 - 10^{100}$ (execute:):

- `1e100 + 0.1 - 1e100` gives `0`;
- **`sum`**`((1e100, 0.1, -1e100))` gives `0`;
- $10^{100} + 0.1 = \underbrace{100\ldots0.1}_{\text{102 digits}}$ has 102 significant decimal digits,
- but **`float`** stores only about 16 (the most significant) of them,
- so `1e100 + 0.1 == 1e100` (check!);
- **`from`** `math` **`import`** `fsum`
  `fsum((1e100, 0.1, -1e100))` gives `0.1`;
- (`1e100 - 1e100 + 0.1` also gives `0.1`).

Exercise: Compare results given by:
`sum``([0.1]*10)` and `fsum([0.1]*10)`

math.fsum – accurate float sum

`math.fsum(iterable)`

Return an accurate **`float`** sum of values in the `iterable`.

Example: let us calculate $10^{100} + 0.1 - 10^{100}$ (execute:):

- `1e100 + 0.1 - 1e100` gives `0`;
- **`sum`**`((1e100, 0.1, -1e100))` gives `0`;
- $10^{100} + 0.1 = \underbrace{100\ldots0.1}_{\text{102 digits}}$ has 102 significant decimal digits,
- but **`float`** stores only about 16 (the most significant) of them,
- so `1e100 + 0.1 == 1e100` (check!);
- **`from`** `math` **`import`** `fsum`
  `fsum((1e100, 0.1, -1e100))` gives `0.1`;
- (`1e100 - 1e100 + 0.1` also gives `0.1`).

Exercise: Compare results given by:
`sum``([0.1]*10)` and `fsum([0.1]*10)`

math.fsum – accurate float sum

`math.fsum(iterable)`

Return an accurate **`float`** sum of values in the `iterable`.

Example: let us calculate $10^{100} + 0.1 - 10^{100}$ (execute:):

- `1e100 + 0.1 - 1e100` gives `0`;
- **`sum`**`((1e100, 0.1, -1e100))` gives `0`;
- $10^{100} + 0.1 = \underbrace{100\ldots0.1}_{\text{102 digits}}$ has 102 significant decimal digits,
- but **`float`** stores only about 16 (the most significant) of them,
- so `1e100 + 0.1 == 1e100` (check!);
- **`from`** `math` **`import`** `fsum`
  `fsum((1e100, 0.1, -1e100))` gives `0.1`;
- (`1e100 - 1e100 + 0.1` also gives `0.1`).

Exercise: Compare results given by:
`sum``([0.1]*10)` and `fsum([0.1]*10)`

math.fsum – accurate float sum

`math.fsum(iterable)`

Return an accurate **`float`** sum of values in the `iterable`.

Example: let us calculate $10^{100} + 0.1 - 10^{100}$ (execute:):

- `1e100 + 0.1 - 1e100` gives `0`;
- **`sum`**`((1e100, 0.1, -1e100))` gives `0`;
- $10^{100} + 0.1 = \underbrace{100\ldots0.1}_{\text{102 digits}}$ has 102 significant decimal digits,
- but **`float`** stores only about 16 (the most significant) of them,
- so `1e100 + 0.1 == 1e100` (check!);
- **`from`** `math` **`import`** `fsum`
  `fsum((1e100, 0.1, -1e100))` gives `0.1`;
- (`1e100 - 1e100 + 0.1` also gives `0.1`).

Exercise: Compare results given by:
`sum``([0.1]*10)` and `fsum([0.1]*10)`

The statistics module

The `statistics` module provides functions for calculating statistics of data, including averages, variance, and standard deviation.

Example: Let us define

```
d = [1.5]*6 + [2.1]*3 + [3]*4
```

and calculate the arithmetic mean of `d`:

```
import statistics as s
s.mean(d)
```

Exercises: calculate (tip: `s.` and `Tab` key or `help(s)` or `s?`):

1. the mode (most common value) of `d`
2. the (population) variance of `d`
3. the (population) standard deviation of `d`
4. the median of `d` (using the "mean of middle two" method)

The statistics module

The `statistics` module provides functions for calculating statistics of data, including averages, variance, and standard deviation.

Example: Let us define

```
d = [1.5]*6 + [2.1]*3 + [3]*4
```

and calculate the arithmetic mean of `d`:

```
import statistics as s
s.mean(d)
```

Exercises: calculate (tip: `s.` and `Tab` key or `help(s)` or `s?`):

1. the mode (most common value) of `d`
2. the (population) variance of `d`
3. the (population) standard deviation of `d`
4. the median of `d` (using the "mean of middle two" method)

The statistics module

The `statistics` module provides functions for calculating statistics of data, including averages, variance, and standard deviation.

Example: Let us define

```
d = [1.5]*6 + [2.1]*3 + [3]*4
```

and calculate the arithmetic mean of `d`:

```
import statistics as s
s.mean(d)
```

Exercises: calculate (tip: `s.` and `Tab` key or `help(s)` or `s?`):

1. the mode (most common value) of `d`
2. the (population) variance of `d`
3. the (population) standard deviation of `d`
4. the median of `d` (using the "mean of middle two" method)

The statistics module

The `statistics` module provides functions for calculating statistics of data, including averages, variance, and standard deviation.

Example: Let us define

```
d = [1.5]*6 + [2.1]*3 + [3]*4
```

and calculate the arithmetic mean of `d`:

```
import statistics as s
s.mean(d)
```

Exercises: calculate (tip: `s.` and `Tab` key or `help(s)` or `s?`):

1. the mode (most common value) of `d` Code: `s.mode(d)`
2. the (population) variance of `d` Code: `s.pvariance(d)`
3. the (population) standard deviation of `d` Code: `s.pstdev(d)`
4. the median of `d` (using the "mean of middle two" method) Code: `m.median(d)`

Generator expressions – syntax

Generator expressions allow for transforming iterable objects (sequences). Their basic syntax is:

```
(expression(v) for v in in_iter)
(expression(v) for v in in_iter if predicate(v))
```

where:

- `expression` – an output expression producing members of the new sequence from members of `in_iter` that (when `if predicate(v)` is present) satisfy `predicate`,
- `in_iter` – the input sequence (iterable object),
- `v` – the variable representing members of `in_iter`,
- `predicate` – expression acting as a filter on members of `in_iter`.

Note that elements of the output sequence are produced lazily (one by one, on demand).

Generator expressions – syntax

Generator expressions allow for transforming iterable objects (sequences). Their basic syntax is:

```
(expression(v) for v in in_iter)
(expression(v) for v in in_iter if predicate(v))
```

where:

- `expression` – an output expression producing members of the new sequence from members of `in_iter` that (when `if predicate(v)` is present) satisfy `predicate`,
- `in_iter` – the input sequence (iterable object),
- `v` – the variable representing members of `in_iter`,
- `predicate` – expression acting as a filter on members of `in_iter`.

Note that elements of the output sequence are produced lazily (one by one, on demand).

Generator expressions – syntax

Generator expressions allow for transforming iterable objects (sequences). Their basic syntax is:

```
(expression(v) for v in in_iter)
(expression(v) for v in in_iter if predicate(v))
```

where:

- `expression` – an output expression producing members of the new sequence from members of `in_iter` that (when `if predicate(v)` is present) satisfy `predicate`,
- `in_iter` – the input sequence (iterable object),
- `v` – the variable representing members of `in_iter`,
- `predicate` – expression acting as a filter on members of `in_iter`.

Note that elements of the output sequence are produced lazily (one by one, on demand).

Generator expressions – syntax

Generator expressions allow for transforming iterable objects (sequences). Their basic syntax is:

```
(expression(v) for v in in_iter)
(expression(v) for v in in_iter if predicate(v))
```

where:

- `expression` – an output expression producing members of the new sequence from members of `in_iter` that (when `if predicate(v)` is present) satisfy `predicate`,
- `in_iter` – the input sequence (iterable object),
- `v` – the variable representing members of `in_iter`,
- `predicate` – expression acting as a filter on members of `in_iter`.

Note that elements of the output sequence are produced lazily (one by one, on demand).

Generator expressions – syntax

Generator expressions allow for transforming iterable objects (sequences). Their basic syntax is:

```
(expression(v) for v in in_iter)
(expression(v) for v in in_iter if predicate(v))
```

where:

- `expression` – an output expression producing members of the new sequence from members of `in_iter` that (when `if predicate(v)` is present) satisfy `predicate`,
- `in_iter` – the input sequence (iterable object),
- `v` – the variable representing members of `in_iter`,
- `predicate` – expression acting as a filter on members of `in_iter`.

Note that elements of the output sequence are produced lazily (one by one, on demand).

Generator expressions – syntax

Generator expressions allow for transforming iterable objects (sequences). Their basic syntax is:

```
(expression(v) for v in in_iter)
(expression(v) for v in in_iter if predicate(v))
```

where:

- `expression` – an output expression producing members of the new sequence from members of `in_iter` that (when `if predicate(v)` is present) satisfy `predicate`,
- `in_iter` – the input sequence (iterable object),
- `v` – the variable representing members of `in_iter`,
- `predicate` – expression acting as a filter on members of `in_iter`.

Note that elements of the output sequence are produced lazily (one by one, on demand).

Generator expressions – syntax

Generator expressions allow for transforming iterable objects (sequences). Their basic syntax is:

```
(expression(v) for v in in_iter)
(expression(v) for v in in_iter if predicate(v))
```

where:

- `expression` – an output expression producing members of the new sequence from members of `in_iter` that (when `if predicate(v)` is present) satisfy `predicate`,
- `in_iter` – the input sequence (iterable object),
- `v` – the variable representing members of `in_iter`,
- `predicate` – expression acting as a filter on members of `in_iter`.

Note that elements of the output sequence are produced lazily (one by one, on demand).

Generator expressions – examples and exercises [1/3]

- `sum((x**2 for x in range(1, 9)))`
  $= \sum_{x=1}^{8} x^2$; the generator is passed to the `sum` function;
- `sum(x**2 for x in range(1, 9))`
  is exactly the same as above;
  double brackets are not necessary if the generator expression is the **sole** argument passed to a function;
- `min(2*a-10 for a in range(1, 8), key=abs)`
  raises `SyntaxError`; since we give an extra (`key`) argument to `min`, the generator expression needs brackets:
- `min((2*a-10 for a in range(1, 8)), key=abs)`
  the element of the set $\{2a - 10; a = 1, 2, \ldots, 7\}$ with the smallest absolute value;

Exercises: calculate:

1. $\sum_{i=0}^{90} 3^i$
2. $\prod_{b=0}^{50} (2^b + b)$

Generator expressions – examples and exercises [1/3]

- `sum((x**2 for x in range(1, 9)))`
  $= \sum_{x=1}^{8} x^2$; the generator is passed to the `sum` function;
- `sum(x**2 for x in range(1, 9))`
  is exactly the same as above;
  double brackets are not necessary if the generator expression is the **sole** argument passed to a function;
- `min(2*a-10 for a in range(1, 8), key=abs)`
  raises `SyntaxError`; since we give an extra (`key`) argument to `min`, the generator expression needs brackets:
- `min((2*a-10 for a in range(1, 8)), key=abs)`
  the element of the set $\{2a - 10; a = 1, 2, \ldots, 7\}$ with the smallest absolute value;

Exercises: calculate:

1. $\sum_{i=0}^{90} 3^i$
2. $\prod_{b=0}^{50} (2^b + b)$

Generator expressions – examples and exercises [1/3]

- `sum((x**2 for x in range(1, 9)))`
  $= \sum_{x=1}^{8} x^2$; the generator is passed to the `sum` function;
- `sum(x**2 for x in range(1, 9))`
  is exactly the same as above;
  double brackets are not necessary if the generator expression is the **sole** argument passed to a function;
- `min(2*a-10 for a in range(1, 8), key=abs)`
  raises `SyntaxError`; since we give an extra (`key`) argument to `min`, the generator expression needs brackets:
- `min((2*a-10 for a in range(1, 8)), key=abs)`
  the element of the set $\{2a - 10; a = 1, 2, \ldots, 7\}$ with the smallest absolute value;

Exercises: calculate:

1. $\sum_{i=0}^{90} 3^i$
2. $\prod_{b=0}^{50} (2^b + b)$

Generator expressions – examples and exercises [1/3]

- `sum((x**2 for x in range(1, 9)))`
  $= \sum_{x=1}^{8} x^2$; the generator is passed to the `sum` function;
- `sum(x**2 for x in range(1, 9))`
  is exactly the same as above;
  double brackets are not necessary if the generator expression is the **sole** argument passed to a function;
- `min(2*a-10 for a in range(1, 8), key=abs)`
  raises `SyntaxError`; since we give an extra (`key`) argument to `min`, the generator expression needs brackets:
- `min((2*a-10 for a in range(1, 8)), key=abs)`
  the element of the set $\{2a - 10; a = 1, 2, \ldots, 7\}$ with the smallest absolute value;

Exercises: calculate:

1. $\sum_{i=0}^{90} 3^i$
2. $\prod_{b=0}^{50} (2^b + b)$

Generator expressions – examples and exercises [1/3]

- `sum((x**2 for x in range(1, 9)))`
  $= \sum_{x=1}^{8} x^2$; the generator is passed to the `sum` function;
- `sum(x**2 for x in range(1, 9))`
  is exactly the same as above;
  double brackets are not necessary if the generator expression is the **sole** argument passed to a function;
- `min(2*a-10 for a in range(1, 8), key=abs)`
  raises `SyntaxError`; since we give an extra (`key`) argument to `min`, the generator expression needs brackets:
- `min((2*a-10 for a in range(1, 8)), key=abs)`
  the element of the set $\{2a - 10; a = 1, 2, \ldots, 7\}$ with the smallest absolute value;

Exercises: calculate:

1. $\sum_{i=0}^{90} 3^i$
2. $\prod_{b=0}^{50} (2^b + b)$

Generator expressions – examples and exercises [1/3]

- `sum((x**2 for x in range(1, 9)))`
  $= \sum_{x=1}^{8} x^2$; the generator is passed to the `sum` function;
- `sum(x**2 for x in range(1, 9))`
  is exactly the same as above;
  double brackets are not necessary if the generator expression is the **sole** argument passed to a function;
- `min(2*a-10 for a in range(1, 8), key=abs)`
  raises `SyntaxError`; since we give an extra (`key`) argument to `min`, the generator expression needs brackets:
- `min((2*a-10 for a in range(1, 8)), key=abs)`
  the element of the set $\{2a - 10; a = 1, 2, \ldots, 7\}$ with the smallest absolute value;

Exercises: calculate:

1. $\sum_{i=0}^{90} 3^i$ Code: `sum(3**i for i in range(91))`
2. $\prod_{b=0}^{50}(2^b + b)$ Code: `from functools import reduce`
   `reduce(lambda x,y: x*y, (2**b+b for b in range(51)))`

Generator expressions – examples and exercises [2/3]

- `sum(x**3 for x in range(2, 79) if x%5 in (2, 4))`
  the sum of x^3 over all $x \in \{2, 3, \ldots, 78\}$ such that the remainder after division of x by 5 is 2 or 4;
- `max(a for a in range(3, 50) if a**3 <= 900)`
  the largest integer $a \in [3, 50)$ satisfying $a^3 \leq 900$;
- `sum(2*x if x%2==0 else x**3 for x in range(9))`

$$= \sum_{x=0}^{8} \begin{cases} 2x, & \text{if x is even,} \\ x^3, & \text{otherwise.} \end{cases}$$

Exercises: calculate:

1. the sum of $5 \cdot b$ over all $b \in \{7, 8, \ldots, 95\}$ such that the remainder after division of b by 17 is in range $(2, 9]$
2. $$\max_{x=0,1,\ldots,7} \begin{cases} \frac{x}{2-x}, & \text{if the remainder after division of x by 3 is 1} \\ x/5, & \text{otherwise} \end{cases}$$

Generator expressions – examples and exercises [2/3]

- `sum(x**3 for x in range(2, 79) if x%5 in (2, 4))`
  the sum of x^3 over all $x \in \{2, 3, \ldots, 78\}$ such that the remainder after division of x by 5 is 2 or 4;
- `max(a for a in range(3, 50) if a**3 <= 900)`
  the largest integer $a \in [3, 50)$ satisfying $a^3 \leq 900$;
- `sum(2*x if x%2==0 else x**3 for x in range(9))`

$$= \sum_{x=0}^{8} \begin{cases} 2x, & \text{if x is even,} \\ x^3, & \text{otherwise.} \end{cases}$$

Exercises: calculate:

1. the sum of $5 \cdot b$ over all $b \in \{7, 8, \ldots, 95\}$ such that the remainder after division of b by 17 is in range $(2, 9]$
2. $$\max_{x=0,1,\ldots,7} \begin{cases} \frac{x}{2-x}, & \text{if the remainder after division of x by 3 is 1} \\ x/5, & \text{otherwise} \end{cases}$$

Generator expressions – examples and exercises [2/3]

- `sum(x**3 for x in range(2, 79) if x%5 in (2, 4))`
  the sum of x^3 over all $x \in \{2, 3, \ldots, 78\}$ such that the remainder after division of x by 5 is 2 or 4;
- `max(a for a in range(3, 50) if a**3 <= 900)`
  the largest integer $a \in [3, 50)$ satisfying $a^3 \leq 900$;
- `sum(2*x if x%2==0 else x**3 for x in range(9))`

$$= \sum_{x=0}^{8} \begin{cases} 2x, & \text{if x is even,} \\ x^3, & \text{otherwise.} \end{cases}$$

Exercises: calculate:

1. the sum of $5 \cdot b$ over all $b \in \{7, 8, \ldots, 95\}$ such that the remainder after division of b by 17 is in range $(2, 9]$
2. $$\max_{x=0,1,\ldots,7} \begin{cases} \frac{x}{2-x}, & \text{if the remainder after division of x by 3 is 1} \\ x/5, & \text{otherwise} \end{cases}$$

Generator expressions – examples and exercises [2/3]

- `sum(x**3 for x in range(2, 79) if x%5 in (2, 4))`
  the sum of x^3 over all $x \in \{2, 3, \ldots, 78\}$ such that the remainder after division of x by 5 is 2 or 4;
- `max(a for a in range(3, 50) if a**3 <= 900)`
  the largest integer $a \in [3, 50)$ satisfying $a^3 \leq 900$;
- `sum(2*x if x%2==0 else x**3 for x in range(9))`

$$= \sum_{x=0}^{8} \begin{cases} 2x, & \text{if x is even,} \\ x^3, & \text{otherwise.} \end{cases}$$

Exercises: calculate:

1. the sum of $5 \cdot b$ over all $b \in \{7, 8, \ldots, 95\}$ such that the remainder after division of b by 17 is in range $(2, 9]$
2. $$\max_{x=0,1,\ldots,7} \begin{cases} \frac{x}{2-x}, & \text{if the remainder after division of x by 3 is 1} \\ x/5, & \text{otherwise} \end{cases}$$

Generator expressions – examples and exercises [2/3]

- `sum(x**3 for x in range(2, 79) if x%5 in (2, 4))`
  the sum of x^3 over all $x \in \{2, 3, \ldots, 78\}$ such that the remainder after division of x by 5 is 2 or 4;
- `max(a for a in range(3, 50) if a**3 <= 900)`
  the largest integer $a \in [3, 50)$ satisfying $a^3 \leq 900$;
- `sum(2*x if x%2==0 else x**3 for x in range(9))`

$$= \sum_{x=0}^{8} \begin{cases} 2x, & \text{if x is even,} \\ x^3, & \text{otherwise.} \end{cases}$$

Exercises: calculate:

1. the sum of $5 \cdot b$ over all $b \in \{7, 8, \ldots, 95\}$ such that the remainder after division of b by 17 is in range $(2, 9]$
   `sum(5*b for b in range(7, 96) if 2 < b%17 <= 9)`
2. $\max\limits_{x=0,1,\ldots,7} \begin{cases} \frac{x}{2-x}, & \text{if the remainder after division of x by 3 is 1} \\ x/5, & \text{otherwise} \end{cases}$
   `max(x/(2-x) if x%3==1 else x/5 for x in range(8))`

Generator expressions – examples and exercises [3/3]

- `sum(x*y for x in range(11) for y in range(11))`
  $= \sum_{x \in \{0,1,\ldots,10\}, y \in \{0,1,\ldots,10\}} xy$;
- `from math import fsum`
  `fsum(x/y for x in range(9) for y in range(1,21))`
  the sum of $\frac{x}{y}$ over all $x = 0, 1, \ldots, 8$ and $y = 1, 2, \ldots, 20$;
  (`fsum` is used instead of `sum` to ensure possibly exact result)
- `from fractions import Fraction as F`
  `sum(F(x,y) for x in range(9) for y in range(1,21))`
  similar to the previous one, but using `Fraction`s.

Exercise: calculate:

1. $\max_{x \in \{0,1,\ldots,5\}, y \in \{1,2,\ldots,7\}} (xy - x^y)$
2. the sum of $\frac{x}{x+y}$ over all $x = 0, 1, \ldots, 7$ and $y = 2, 4, 7, 8$
   (give 2 solutions, using: `float`s+`fsum` and `Fraction`s+`sum`)

Generator expressions – examples and exercises [3/3]

- `sum(x*y for x in range(11) for y in range(11))`
  $= \sum_{x \in \{0,1,\ldots,10\}, y \in \{0,1,\ldots,10\}} xy$;
- `from math import fsum`
  `fsum(x/y for x in range(9) for y in range(1,21))`
  the sum of $\frac{x}{y}$ over all $x = 0, 1, \ldots, 8$ and $y = 1, 2, \ldots, 20$;
  (`fsum` is used instead of `sum` to ensure possibly exact result)
- `from fractions import Fraction as F`
  `sum(F(x,y) for x in range(9) for y in range(1,21))`
  similar to the previous one, but using `Fraction`s.

Exercise: calculate:

1. $\max_{x \in \{0,1,\ldots,5\}, y \in \{1,2,\ldots,7\}} (xy - x^y)$
2. the sum of $\frac{x}{x+y}$ over all $x = 0, 1, \ldots, 7$ and $y = 2, 4, 7, 8$
   (give 2 solutions, using: `floats`+`fsum` and `Fraction`s+`sum`)

Generator expressions – examples and exercises [3/3]

- `sum(x*y for x in range(11) for y in range(11))`
  $= \sum_{x \in \{0,1,\ldots,10\}, y \in \{0,1,\ldots,10\}} xy$;
- `from math import fsum`
  `fsum(x/y for x in range(9) for y in range(1,21))`
  the sum of $\frac{x}{y}$ over all $x = 0, 1, \ldots, 8$ and $y = 1, 2, \ldots, 20$;
  (`fsum` is used instead of `sum` to ensure possibly exact result)
- `from fractions import Fraction as F`
  `sum(F(x,y) for x in range(9) for y in range(1,21))`
  similar to the previous one, but using `Fraction`s.

Exercise: calculate:

1. $\max_{x \in \{0,1,\ldots,5\}, y \in \{1,2,\ldots,7\}} (xy - x^y)$
2. the sum of $\frac{x}{x+y}$ over all $x = 0, 1, \ldots, 7$ and $y = 2, 4, 7, 8$
   (give 2 solutions, using: `floats`+`fsum` and `Fraction`s+`sum`)

Generator expressions – examples and exercises [3/3]

- `sum(x*y for x in range(11) for y in range(11))`
  $= \sum_{x \in \{0,1,\ldots,10\}, y \in \{0,1,\ldots,10\}} xy$;
- `from math import fsum`
  `fsum(x/y for x in range(9) for y in range(1,21))`
  the sum of $\frac{x}{y}$ over all $x = 0, 1, \ldots, 8$ and $y = 1, 2, \ldots, 20$;
  (`fsum` is used instead of `sum` to ensure possibly exact result)
- `from fractions import Fraction as F`
  `sum(F(x,y) for x in range(9) for y in range(1,21))`
  similar to the previous one, but using `Fraction`s.

Exercise: calculate:

1. $\max_{x \in \{0,1,\ldots,5\}, y \in \{1,2,\ldots,7\}} (xy - x^y)$

2. the sum of $\frac{x}{x+y}$ over all $x = 0, 1, \ldots, 7$ and $y = 2, 4, 7, 8$
   (give 2 solutions, using: `float`s+`fsum` and `Fraction`s+`sum`)

Generator expressions – examples and exercises [3/3]

- `sum(x*y for x in range(11) for y in range(11))`
  $= \sum_{x\in\{0,1,\ldots,10\},y\in\{0,1,\ldots,10\}} xy$;
- `from math import fsum`
  `fsum(x/y for x in range(9) for y in range(1,21))`
  the sum of $\frac{x}{y}$ over all $x = 0, 1, \ldots, 8$ and $y = 1, 2, \ldots, 20$;
  (`fsum` is used instead of `sum` to ensure possibly exact result)
- `from fractions import Fraction as F`
  `sum(F(x,y) for x in range(9) for y in range(1,21))`
  similar to the previous one, but using `Fraction`s.

Exercise: calculate:

1. $\max_{x\in\{0,1,\ldots,5\},y\in\{1,2,\ldots,7\}}(xy - x^y)$
   `max(x*y-x**y for x in range(6) for y in range(1,8))`
2. the sum of $\frac{x}{x+y}$ over all $x = 0, 1, \ldots, 7$ and $y = 2, 4, 7, 8$
   (give 2 solutions, using: `floats`+`fsum` and `Fraction`s+`sum`)
   `fsum(x/(x+y) for x in range(8) for y in (2,4,7,8))`
   `sum(F(x,x+y) for x in range(8) for y in (2,4,7,8))`

Generator expressions – constructing lists and tuples

Generator expressions can be used for constructing `lists` or `tuples`. For example:

- `list(x**2 for x in range(1, 5))`
  constructs the `list` `[1**2, 2**2, 3**2, 4**2]`;
- `[x**2 for x in range(1, 5)]`
  is a *list comprehension* and gives the same `list` as above;
- `tuple(x**2 for x in range(1, 5))`
  constructs the `tuple` `(1**2, 2**2, 3**2, 4**2)`;

However:

- `(x**2 for x in range(1, 5))`
  is not a `tuple` but a generator, try:
  `a=(x**2 for x in range(1, 5))`
  `type(a)`
  (brackets cannot be omitted in the line `a=(...)`)
- `[(x**2 for x in range(1, 5))]`
  is not a *list comprehension* but a one-element `list` that includes a generator.

Generator expressions – constructing lists and tuples

Generator expressions can be used for constructing **lists** or **tuples**. For example:

- `list(x**2 for x in range(1, 5))`
  constructs the **list** `[1**2, 2**2, 3**2, 4**2]`;
- `[x**2 for x in range(1, 5)]`
  is a *list comprehension* and gives the same **list** as above;
- `tuple(x**2 for x in range(1, 5))`
  constructs the **tuple** `(1**2, 2**2, 3**2, 4**2)`;

However:

- `(x**2 for x in range(1, 5))`
  is not a **tuple** but a generator, try:
  `a=(x**2 for x in range(1, 5))`
  `type(a)`
  (brackets cannot be omitted in the line `a=(...)`)
- `[(x**2 for x in range(1, 5))]`
  is not a *list comprehension* but a one-element **list** that includes a generator.

Generator expressions – constructing lists and tuples

Generator expressions can be used for constructing **lists** or **tuples**. For example:

- `list(x**2 for x in range(1, 5))`
  constructs the **list** `[1**2, 2**2, 3**2, 4**2]`;
- `[x**2 for x in range(1, 5)]`
  is a *list comprehension* and gives the same **list** as above;
- `tuple(x**2 for x in range(1, 5))`
  constructs the **tuple** `(1**2, 2**2, 3**2, 4**2)`;

However:

- `(x**2 for x in range(1, 5))`
  is not a **tuple** but a generator, try:
  `a=(x**2 for x in range(1, 5))`
  `type(a)`
  (brackets cannot be omitted in the line `a=(...)`)
- `[(x**2 for x in range(1, 5))]`
  is not a *list comprehension* but a one-element **list** that includes a generator.

Generator expressions – constructing lists and tuples

Generator expressions can be used for constructing **lists** or **tuples**. For example:

- `list(x**2 for x in range(1, 5))`
  constructs the **list** `[1**2, 2**2, 3**2, 4**2]`;
- `[x**2 for x in range(1, 5)]`
  is a *list comprehension* and gives the same **list** as above;
- `tuple(x**2 for x in range(1, 5))`
  constructs the **tuple** `(1**2, 2**2, 3**2, 4**2)`;

However:

- `(x**2 for x in range(1, 5))`
  is not a **tuple** but a generator, try:
  `a=(x**2 for x in range(1, 5))`
  `type(a)`
  (brackets cannot be omitted in the line `a=(...)`)
- `[(x**2 for x in range(1, 5))]`
  is not a *list comprehension* but a one-element **list** that includes a generator.

Generator expressions – constructing lists and tuples

Generator expressions can be used for constructing **lists** or **tuples**. For example:

- `list(x**2 for x in range(1, 5))`
  constructs the **list** `[1**2, 2**2, 3**2, 4**2]`;
- `[x**2 for x in range(1, 5)]`
  is a *list comprehension* and gives the same **list** as above;
- `tuple(x**2 for x in range(1, 5))`
  constructs the **tuple** `(1**2, 2**2, 3**2, 4**2)`;

However:

- `(x**2 for x in range(1, 5))`
  is not a **tuple** but a generator, try:
  `a=(x**2 for x in range(1, 5))`
  `type(a)`
  (brackets cannot be omitted in the line `a=(...)`)
- `[(x**2 for x in range(1, 5))]`
  is not a *list comprehension* but a one-element **list** that includes a generator.

Generator expressions – constructing lists and tuples

Generator expressions can be used for constructing **lists** or **tuples**. For example:

- `list(x**2 for x in range(1, 5))`
  constructs the **list** `[1**2, 2**2, 3**2, 4**2]`;
- `[x**2 for x in range(1, 5)]`
  is a *list comprehension* and gives the same **list** as above;
- `tuple(x**2 for x in range(1, 5))`
  constructs the **tuple** `(1**2, 2**2, 3**2, 4**2)`;

However:

- `(x**2 for x in range(1, 5))`
  is not a **tuple** but a generator, try:
  ```
  a=(x**2 for x in range(1, 5))
  type(a)
  ```
  (brackets cannot be omitted in the line `a=(...)`)
- `[(x**2 for x in range(1, 5))]`
  is not a *list comprehension* but a one-element **list** that includes a generator.

Generator expressions and lists [1/2]

Let `a` and `b` be **`lists`** of 100 random **`floats`** in the range $[0, 5]$:

```
import random
a = [random.uniform(0, 5) for _ in range(100)]
b = [random.uniform(0, 5) for _ in range(100)]
```

Example: `fsum(x*y for x in a for y in b if x<y)`
is the sum of $x \cdot y$ over all $x \in a$, $y \in b$ such that $x < y$.

Exercises: calculate:

1. the sum of x^y over all $x \in a$, $y \in b$ such that $x > y$
2. whether $x < 4.9$ for all $x \in a$ (hint: see **`all`**?)
3. median of v^3 over all $v \in b$ such that $v \geq min(a)$

Generator expressions and lists [1/2]

Let `a` and `b` be **`lists`** of 100 random **`floats`** in the range $[0, 5]$:

```
import random
a = [random.uniform(0, 5) for _ in range(100)]
b = [random.uniform(0, 5) for _ in range(100)]
```

Example: `fsum(x*y for x in a for y in b if x<y)`
is the sum of $x \cdot y$ over all $x \in a$, $y \in b$ such that $x < y$.

Exercises: calculate:

1. the sum of x^y over all $x \in a$, $y \in b$ such that $x > y$
2. whether $x < 4.9$ for all $x \in a$ (hint: see `all?`)
3. median of v^3 over all $v \in b$ such that $v \geq min(a)$

Generator expressions and lists [1/2]

Let `a` and `b` be **`lists`** of 100 random **`floats`** in the range $[0, 5]$:

```
import random
a = [random.uniform(0, 5) for _ in range(100)]
b = [random.uniform(0, 5) for _ in range(100)]
```

Example: `fsum(x*y for x in a for y in b if x<y)`
is the sum of $x \cdot y$ over all $x \in a$, $y \in b$ such that $x < y$.

Exercises: calculate:

1. the sum of x^y over all $x \in a$, $y \in b$ such that $x > y$
2. whether $x < 4.9$ for all $x \in a$ (hint: see **`all`**`?`)
3. median of v^3 over all $v \in b$ such that $v \geq min(a)$

Generator expressions and lists [1/2]

Let `a` and `b` be **`lists`** of 100 random **`floats`** in the range $[0, 5]$:

```
import random
a = [random.uniform(0, 5) for _ in range(100)]
b = [random.uniform(0, 5) for _ in range(100)]
```

Example: `fsum(x*y for x in a for y in b if x<y)` is the sum of $x \cdot y$ over all $x \in a$, $y \in b$ such that $x < y$.

Exercises: calculate:

1. the sum of x^y over all $x \in a$, $y \in b$ such that $x > y$
   ```
   fsum(x**y for x in a for y in b if x>y)
   ```
2. whether $x < 4.9$ for all $x \in a$ (hint: see **`all`**`?`)
   ```
   all(x<4.9 for x in a)
   ```
3. median of v^3 over all $v \in b$ such that $v \geq min(a)$
   ```
   from statistics import median
   min_a = min(a)
   median(v**3 for v in b if v >= min_a)
   ```

Generator expressions and lists [2/2]

1. `a[0]·b[0] + a[1]·b[1] + ··· + a[99]·b[99]`
   hint: see **`zip`**`?`; **`zip`**`(a,b)` gives the sequence of **`tuple`**`s`:
   `(a[0],b[0])` , `(a[1],b[1])`, . . . , `(a[99],b[99])`;

2. the sum of $\sin(x)$ over all x in the concatenation of `a` and `b`;

3. whether $|v - 2| \in [0.9, 1]$ for any $v \in b$ (hint: see **`any`**`?`).

Generator expressions and lists [2/2]

1. `a[0]·b[0] + a[1]·b[1] +···+ a[99]·b[99]`
   hint: see **`zip`**?; **`zip`**`(a,b)` gives the sequence of **`tuples`**:
   `(a[0],b[0])` , `(a[1],b[1])`,..., `(a[99],b[99])`;
   `fsum(x*y for x, y in zip(a, b))` works even if `a` and `b` are not random access (have no `[]` operator and can produce items only one by one) and so is more general than
   `fsum(a[i]*b[i] for i in range(100))`
2. the sum of $\sin(x)$ over all x in the concatenation of `a` and `b`;
   ```
   from math import sin
   fsum(sin(x) for x in a+b)
   ```
   creates `a+b` as a **`list`** and may use more memory than:
   ```
   from itertools import chain
   fsum(sin(x) for x in chain(a, b))
   ```
   (see `chain?` or **`help`**`(chain)`)
3. whether $|v-2| \in [0.9, 1]$ for any $v \in b$ (hint: see **`any`**?).
   ```
   any(0.9 <= abs(v-2) <= 1 for v in b)
   ```

Homework

Create the `list` L of 40 random `float`s in the range $[0, 9]$ and calculate:

1. the arithmetic mean, the mode, and the median of L;
2. whether $x^2 \in [2, 5)$ for any $x \in L$;
3. $\prod_{x \in L} (x \cdot 2^{5x}) - \min_{v \in L} \begin{cases} v - 4 + 3\sin^2(v), & \text{if } v > 5, \\ v^2 + 5\cos^3(v), & \text{otherwise;} \end{cases}$
4. the member of $\{x - 8; x \in L\}$ with the largest absolute value;
5. $\max_{i=0,\ldots,39}(i \cdot L[i] - i)$; use `enumerate` to solve the task without using `L`'s `[]` operator, as if `L` was not random access;
6. the sum of $\frac{a}{b}$ over all $a \in L$, $b \in \{2, 3, 8, 9\}$ such that $a^2 \geq b$;
7. (hard, non-mandatory) which of the sets $[i, i + 1)$ for $i \in \{0, 1 \ldots, 8\}$ contains the largest number of L members.

Calculate $\sum_{d=1}^{10000} \frac{1}{d}$. Compare results (as `float`s) and speeds of 3 solutions which use: `float`+`sum`, `float`+`fsum`, `Fraction`+`sum`.

Please note all the expressions used and the results obtained.

Bibliography

- Official *Python documentation* available on https://docs.python.org/3/, topics: *More Control Flow Tools / The range() Function*, *Classes / Generator Expressions*, and *Data Structures / More on Lists / List Comprehensions* (in *The Python Tutorial*), *Sequence Types – list, tuple, range* (in *Library Reference / Built-in Types*), modules: *statistics*, *functools*, *itertools*, *math*
- Guido van Rossum *From List Comprehensions to Generator Expressions*, available on http://python-history.blogspot.com/...
- Jakub Przywóski *Python Reference (The Right Way)* available on http://python-reference.readthedocs.io/, topic: *Comprehensions and Generator Expression*
- *List comprehension* in *Wikipedia, the free encyclopedia*, available on https://en.wikipedia.org/wiki/List_comprehension