Special methods for numeric types

- If you want the instances of your class to support the numeric operations (like +, −, *, /, etc), you must define a set of special methods in your class.
- For example, the expression \( x+y \) will result in a call \( x.__add__(y) \) which should return the result of the operation.
- Here are a few of the most common numerical special methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>Addition (+)</td>
</tr>
<tr>
<td><strong>sub</strong></td>
<td>Subtraction (-)</td>
</tr>
<tr>
<td><strong>mul</strong></td>
<td>Multiplication (*)</td>
</tr>
<tr>
<td><strong>div</strong></td>
<td>Division (/)</td>
</tr>
</tbody>
</table>
The corresponding augmented assignments `+= -= *= /=` have special methods `__iadd__`, `__isub__`, `__imul__`, `__idiv__`

The conversion functions `complex()`, `float()`, `int()` and `long()` call the following special methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__complex__</code></td>
<td>convert to a complex number</td>
</tr>
<tr>
<td><code>__float__</code></td>
<td>convert to a float</td>
</tr>
<tr>
<td><code>__int__</code></td>
<td>convert to an integer</td>
</tr>
<tr>
<td><code>__long__</code></td>
<td>convert to a long</td>
</tr>
</tbody>
</table>
Special methods for containers 1

- In addition to the normal methods of containers, like the `append` method of the list, there are several operations that are handled by calls to special methods of the container class.
- The test whether \( x \) is a member of container \( c \) is done by the operation \( x \text{ in } c \). The corresponding special method call is \( x.__contains__(y) \).
- Deletion of an element of container \( c \) can be done with the operation \( \text{del } c[\text{key}] \). This will result in the method call \( x.__delitem__ \).
Objects and classes

Special methods for containers 2

- Reading an item of a container $c$ is done with the operation $c[key]$. The corresponding method call is $c.__getitem__(key)$
- Similarly, setting an item with $c[key]=value$ results in the call $c.__setitem__(key,value)$
- The number of elements in a container $c$ can be queried with the function call $\text{len}(c)$. This function call actually calls the special method $c.__len__$
- The call $\text{iter}(c)$ will call the special method $\text{__iter__}$. More about the purpose of this function in the next few slides.
Iterators 1

- *Iterator* is an interface that allows us to access consequently the elements of any *logical* or *physical sequence*

- A list is an example of a physical sequence: its elements are stored in the memory

- A range `xrange(0,100)` is an example of a logical sequence, only its end points are stored.

- All iterators have a method called `next` that returns an element of the sequence and moves forward one position in the sequence
What happens when we have already traversed through the elements of a sequence and we call the `next` method?

- Trying to pass the end of sequence causes an exception `StopIteration` to be dispatched.
- For example, the `for` loop asks for new elements with the `next` method until it notices this exception. Then it stops the loop.
- The `try/except` compound statement can be used to detect the exception.
How the for loop does the iteration

- The following two pieces of code are equivalent
- Loop done with for:
  ```python
  for x in c:
    # statements
  ```
- Loop done with an iterator:
  ```python
  it = iter(c)
  while True:
    try:
      x = it.next()
    except StopIteration:
      break
    # statements
  ```
Where do iterators come from? 1

- An *iterable* is an object that either is an iterator or that can create an iterator if asked
- We can ask for an iterator from an object \(x\) with the function call `iter(x)`
- The `iter` function usually just calls the special method \(x.__iter__()\) of \(x\)
- For example, if we have a list \(L=[1,2,3]\), we can request an iterator from it with the call `L.__iter__()` or preferably with `iter(L)`
Where do iterators come from? 2

- If we have an iterator called it, the call iter(it) should return the same iterator unmodified.

- So, it doesn’t hurt if we call iter more than once. For instance, the loop "for i in x" can call iter(x) no matter whether x is a sequence or an iterator.

- All sequences (like str, tuple and list) are iterable. Also sets and dictionaries are iterable.
Let’s say we want to create an iterator that yields all the triples \((0,0,0), (0,0,1), \ldots, (0,0,9), \ldots (9,9,9)\) (there are one thousand of these triples).

We can easily go through these triples with nested `for` loops as follows:

```python
for i in range(10):
    for j in range(10):
        for k in range(10):
            print (i, j, k)
```

But to implement this so that the `next` method of the iterator always returns one element at a time, cannot be done with nested loops in a normal way. The result would be very complicated.
Generators provide a solution to this

- If a function definition contains the statement `yield`, then function behaves in a special way
- When we call this function, it doesn’t execute the function body.
- Instead, when called, the function returns an iterator generated by the function
- An example:

```python
def triples():
    for i in range(10):
        for j in range(10):
            for k in range(10):
                yield (i,j,k)
```
Generators and generator expressions

- When the `next` method of this special iterator is called, the function executes until it meets the `yield` statement and returns the corresponding `yield` value.
- When we call `next` again it continues where the previous `yield` left.
- This functionality is even easier to achieve with the generator expressions we saw in the basics section of this course:

```python
( (i,j,k) for i in range(10) 
   for j in range(10) 
   for k in range(10) )
```
When trying to understand the structure of objects and their relationships, the following functions can be useful:

- `type()`
- `dir()`
- `help()`
- `isinstance()`
- `issubclass()`
- `id()`
Introspection 2

- For each object of Python there is a unique *idnumber* (usually its memory address) that can be queried with the `id()` function

- The operator `is` can be used to ask whether two names refer to the same object. We have the equivalence `x is y` is true if and only if `id(x) == id(y)`

- Let’s have the following iterator example: `x=iter([1,2,3])` and `y=iter(x)`. Since `iter` is an idempotent function, we have `x is y`. 