General

Custom comparison operators

To explain how Python must compare two objects of a self-defined data type (*class*), a specific implementation for the comparison operators must be provided. This can be done by overloading the following magical methods:

method	operator
`__lt__`	$<$
`__le__`	$\leq$
`__gt__`	$>$
`__ge__`	$\geq$
`__eq__`	$=$
`__ne__`	$\neq$

Please note that in most cases you'll have the opportunity to define most of these comparison operators based on the other comparison operators. For example, two object are different if they are not equal.

The format specifier `!r`

Python has two built-in functions that can be used to convert an object into a string: `str()` and `repr()`. By default, Python uses the built-in function `str()` to convert an object to a string in an f-string and when using the string method `str.format()`. If you want to use the built-in function `repr()` instead, you can either explicitly call the function or use the *format specifier* `!r`.

```
>>> course = 'programmming'
>>> str(course)                                    # explicit call to str()
'programmming'
>>> repr(course)                                   # explicit call to repr()
"'programmming'"
>>> course                                         # implicit call to repr()
'programmming'
>>> print(course)                                  # implicit call to str()
programmming
>>> f'The name of the course is {course}.'         # implicit call to str()
'The name of the course is programmming.'
>>> f'The name of the course is {repr(course)}.'   # explicit call to repr()
"The name of the course is 'programmming'."
>>> f'The name of the course is {course!r}.'       # implicit call to repr()
"The name of the course is 'programmming'."
```

Operator overloading with custom types

If Python needs to evaluate the following expression

```
o1 + o2
```

it converts the expression into

```
type(o1).__add__(o1, o2)
```

This way, you can specify how the `+`-operator is evaluated if the object `o1` belongs to a custom type (defined using the `class` keyword). This is called *operator overloading*. However, operator overloading is not restricted to the `+`-operator. In fact, Python converts each built-in operator (like mathematical operators and comparison operators) into calling a method on the left operand `o1`, whose name has been fixed by the Python developers

(all names begin and end with a double underscore). Here's an overview of some of these *magic* methods that correspond to operators:

operator	method
`+`	`__add__`
`-`	`__sub__`
`*`	`__mul__`
`/`	`__truediv__`
`//`	`__floordiv__`
`**`	`__pow__`

Operator overloading initially converts the evaluation of an operator into calling a *magic* method on the left operand `o1`. But what if the class of the left operand `o1` does not define the magic method for objects of `type(o2)`? In that case an exception is raised, and Python makes a second attempt to call another *magic* method (whose name has an extra letter `r` in front) on the right operand `o2`.

For example, if the addition we observed above fails when calling the `__add__` method on the left operand `o1`, Python attempts to call the following method on the right operand `o2`

```
type(o2).__radd__(o2, o1)
```

Note that the name of the method has become `__radd__` instead of `__add__`, and that the order of the arguments has been inverted. This is important for asymmetric operations.

Returning a reference to the current object

Some methods need to return a reference to the object on which they were called. This object is automatically passed as the first argument to the method, and so it is assigned to the parameter `self` if the Python naming convention for the first parameter of a method was followed.

Say we want to implement the Tic-Tac-Toe game:

```
class TicTacToe:

  def __init__(self):
    self.grid = [
      [ None, None, None ],
      [ None, None, None ],
      [ None, None, None ]
    ]
    self.player = 'O'

  def play(self, i, j):
    self.grid[i][j] = self.player
    self.player = 'O' if self.player == 'X' else 'X'
    return self
```

This allows us to play the game in the following

```
>>> game = TicTacToe().play(1, 1).play(0, 0).play(0, 1).play(1, 0)
>>> game.grid
[
  ['X', 'O', None],
  ['X', 'O', None],
  [None, None, None]
]
```

where we can chain multiple calls of the method `TicTacToe.play()` because each method call returns a reference to the object on which it was called: `return self`.

Using `self`

If you work with classes, you need to make a distinction between two kinds of variables. There are **object properties** that can be referenced in all methods of the class and there are **local variables** of methods that are only accessible in the method where they are defined. Only the names of the object properties need to be prefixed with `self`. Variables that are local to a method (local variables) do not need to be prefixed with `self`, and its considered very bad programming style if you do so.

Initialize object properties in initialization method

Before you start with the implementation of a class, you must first determine which properties the objects of the class will have. These variables describe the internal state of the individual objects and can be addressed in all methods of the class. Object properties can be referenced by putting the prefix `self.` in front of their name. It's always a good idea to define object properties in the `__init__` method, where you assign them an initial value.