General

Gebruik van `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 class methods and there are local variables of methods that are only acccessible 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 (the 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 intialisation method

Before you start with the implementation of a class, you must first determine which properties the objects of the class will have. Each of these properties will correspond to a variable that is prefixed with `self.`. These variables describe the internal state of the individual objects and can be addressed in all class methods. It's always a good idea to define the object properties in the `__init__` method, where you assign them an initial value.

Rijksregisternummer

Check data types with `isinstance`

To check whether or not a given object `o` has a given data type `t`, you can use the built-in function `type(o)` to see if it returns the data type `t` for the object `o`. However, it is better (more pythonic) to use the built-in function `isinstance(o, t)` in this case. This function returns a Boolean value that indicates whether or not the object `o` has data type `t` or a date type that is derived from the data type `t`.

```
>>> type(3) == int
True
>>> isinstance(3.14, int)
False
>>> isinstance(3.14, float)
True
>>> isinstance([1, 2, 3], list)
True
```

To check whether or not a given object `o` has one of multiple types, the following syntax can be used. All posible valid types are hereby listed in a tuple.

```
>>> isinstance(3, (str, int))
True
>>> isinstance('a', (str, int))
True
>> isinstance(['a'], (str, int))
False
```

Mad Libs

String method `capitalize()`

The string method `capitalize` can be used to convert the first character of a string to uppercase (if it is a letter) and all other characters to lowercase (if they are letters).

```
>>> 'aBcD'.capitalize()
'Abcd'
```

Specific information

To implement the method `fill` the `split` function can be used. When executing `text.split('_')` with the give text a list will be returned with alternately a piece of text that does not to be substituded and a piece of text that has to be substituted.

```
>>> tekst = '_Name_ created _thing_ so that _CITIZENS_ would learn _discipline_.'
>>> tekst.split('_')
['', 'Name', ' created ', 'thing', ' so that ', 'CITIZENS', ' would learn ', 'discipline', '.']
```

Quipu

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 begins and ends with a double underscore). Here's an overview of some of these *magic* methods:

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 object of type

`o2`? In that case an exception is thrown, 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.