Advanced Python usage: magic methods

Object oriented programming

Python built-in methods

- Everything is an object.
- The base type (class) in Python is `object`.
- Python defines several built-in methods (enclosed by `__`: the "dunder" - double underscore)
- `__init__` is the constructor (initializes values)
- There are other ones:
  - `__new__` (the actual "constructor", outside the scope of this course)
  - `__str__`: returns a string representation of the object

> The dunder methods are "magic" because they are run without being called explicitly.

Example: the `Person` class

```python
class Person:
    def __init__(self, name):
        self._name = name
    def __str__(self):
        return f"Person: {self._name}"
```

- `__str__` will be called when casting the object into a string (for example, in a `print` statement, or when using `str(my_instance)`).

```python
tim = Person("Tim")

print(tim)
# Also possible: str(tim)
```

Other useful methods to override

- `instance` is an instance of the class considered

Method	Goal
`__str__(self)`	For `print(instance)`, or `str(instance)`
`__len__(self)`	Allows `len(instance)`
`__getitem__(self, key)`	Allows `instance[value]` (can be a slice)
`__call__(self)`	Allows to use `instance()`
`__contains__(self, other)`	Allows to test `something in instance` expression
`__iter__(self)`	Allows to use `instance` as an iterator
`__next__(self)`	Returns the next value in the iteration (see above)

Dunder "mathematical" methods

Method	Goal
`__add__(self, right)`	Using `instance + something`
`__sub__(self, right)`	Using `instance - something`
`__mul__(self, right)`	Using `instance * something`
`__pow__(self, right)`	Using `instance ** something`
`__mod__(self, right)`	Using `instance % something`
`__eq__(self, right)`	Using `instance == something`
`__lt__(self, right)`	Using `instance < something`. Needed for sorting!
`__le__(self, right)`	Using `instance <= something`
`__gt__(self, right)`	Using `instance > something`

- See the official documentation for more info.

Implementing the `__lt__` method for the Score class

```python
class Score:
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def __lt__(self, other):
        if type(other) is not type(self):
            raise TypeError("Unsupported type")
        # We sort on the score
        return self.score < other.score
```

- Sort list of scores:

```python
scores = [Score("Tim", 0), Score("John", 1000), Score("Sarah", 2000)]
print(sorted(scores))   # Will display the sorted result
scores.sort()           # Will sort in place
```

Design pattern: using collections

The high scores board is a *collection* of scores.

```python
class HighScores:
    def __init__(self):
        self._scores = list()
        # We would need to manage scores here, or use aggregation
    def __len__(self):
        return len(self._scores)
```

And then:

```python
hiscores = HighScores()
hiscores.add(tim_score)
len(team)        # Will return 1 (1 score in the list)
```

Using `operator` for sorting (and other things)

- The standard library comes with the `operator` module.
- Provides premade functions to access items / elements of an object / collection
- `itemgetter`: to get items from lists/dictionaries by key
- `attrgetter`: to get attributes from objects (by name)

```
menu = [
    ("Pizza", 10),
    ("Pizza slice", 3),
    ("Fountain drink", 2),
    ("Cookie", 4),
]

# Each element in the menu is a tuple (~list)
# We want to sort on the item with index 1
sorted(menu, key=operator.itemgetter(1))
```

With dictionaries

```
menu = [
    { "name": "pizza", "price": 10, "in stock": 10 },
    { "name": "drink", "price": 2, "in stock": 50 },
    { "name": "cookie", "price": 4, "in stock": 20 },
    { "name": "pizza slice", "price": 3, "in stock": 15 },
]
sorted(menu, key=operator.itemgetter('price'))
```

With objects

```
hiscores = [
    Score(name="Tim", score=20),
    Score(name="John", score=0),
    Score(name="Sarah", score=100),
]
# Sorting on the name attribute
sorted(hiscores, key=operator.attrgetter('name'))
```

Collections: getting elements with `__getitem__`

```python
class HighScores:
    def __init__(self):
        # This is a list of scores
        self._scores = list()

    def __len__(self):
        return len(self._scores)

    def __getitem__(self, idx):
        return self._scores[idx]

hiscores = HighScores()
# Add scores to the instance, and then:
print(hiscores[0].scores)     # First score in the list
print(hiscores[-1].scores)    # Last score in the list
```