PROGRAMMING IN PYTHON I

Classes: Advanced Topics

Andreas Schörgenhumer

Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Institute for
Machine Learning

Copyright Statement

Contact

Andreas Schörgenhumer

Institute for Machine Learning
Johannes Kepler University
Altenberger Str. 69
A-4040 Linz

E-Mail: schoergenhumer@ml.jku.at

Write mails only for personal questions

Institute ML Homepage

ROOT CLASS OBJECT

Root Class object

- Every class in Python extends the root/base class **object**.

Root Class object

- Every class in Python extends the root/base class **object**.
- This means that we inherit everything that is available in `object`, which are several useful **special methods**:[1]
  - `__eq__(self, other)`
  - `__hash__(self)`
  - `__str__(self)`
  - `__lt__(self, other)`
  - `...`

[1] No attributes are inherited because `object` does not have any (instance) attributes.

Root Class object

- Every class in Python extends the root/base class **object**.
- This means that we inherit everything that is available in `object`, which are several useful **special methods**:[1]
  - `__eq__(self, other)`
  - `__hash__(self)`
  - `__str__(self)`
  - `__lt__(self, other)`
  - `...`
- Examples and explanations (for some) are part of the supplementary code file.

[1] No attributes are inherited because `object` does not have any (instance) attributes.

Example: Overriding Equality Comparison

- Often, it would be nice to check the instances of our custom classes for equality using the == operator, e.g.:

```
class Dog:
    # Implementation

dog1 = Dog(...)
dog2 = Dog(...)
if dog1 == dog2:
    # Do something
```

Example: Overriding Equality Comparison

- Often, it would be nice to check the instances of our custom classes for equality using the == operator, e.g.:

```
class Dog:
    # Implementation

dog1 = Dog(...)
dog2 = Dog(...)
if dog1 == dog2:
    # Do something
```

- By default, this equality comparison falls back to a reference/identity check (`dog1 is dog2`).

Example: Overriding Equality Comparison

- Often, it would be nice to check the instances of our custom classes for equality using the == operator, e.g.:

```
class Dog:
    # Implementation

dog1 = Dog(...)
dog2 = Dog(...)
if dog1 == dog2:
    # Do something
```

- By default, this equality comparison falls back to a reference/identity check (`dog1 is dog2`).
- This is the default behavior of the special `object` method `__eq__(self, other)`, which will be automatically invoked when the == operator is used.

Example: Overriding Equality Comparison

- We can **override** this special method to provide custom behavior for our `Dog` objects!

```
class Dog:
    def __eq__(self, other):
        if isinstance(other, Dog):
            # Whatever checks we need to perform
            return self.name == other.name and ...
        return NotImplemented  # See code file
```

Example: Overriding Equality Comparison

- We can **override** this special method to provide custom behavior for our `Dog` objects!

```
class Dog:
    def __eq__(self, other):
        if isinstance(other, Dog):
            # Whatever checks we need to perform
            return self.name == other.name and ...
        return NotImplemented  # See code file
```

- Now, whenever something like `some_dog == x` is encountered, this is automatically translated into `Dog.__eq__(self=some_dog, other=x)`.

Example: Overriding Equality Comparison

- We can **override** this special method to provide custom behavior for our `Dog` objects!

```
class Dog:
    def __eq__(self, other):
        if isinstance(other, Dog):
            # Whatever checks we need to perform
            return self.name == other.name and ...
        return NotImplemented  # See code file
```

- Now, whenever something like `some_dog == x` is encountered, this is automatically translated into `Dog.__eq__(self=some_dog, other=x)`.
- This is also called **operator overloading**, since we changed the behavior of our `Dog`'s `==` operator.

MORE OPERATOR OVERLOADING

More Operator Overloading

- There are also some operators we can overload that are not part of `object`.

More Operator Overloading

- There are also some operators we can overload that are not part of `object`.
- This means that there is no default support, but we can easily enable it and provide support ourselves.

More Operator Overloading

- There are also some operators we can overload that are not part of `object`.
- This means that there is no default support, but we can easily enable it and provide support ourselves.
- Some examples:
  - add **+**: `__add__(self, other)`
  - multiply *****: `__mul__(self, other)`
  - get-indexing **[]**: `__getitem__(self, key)`
  - contains **in**: `__contains__(self, item)`
  - ...

ENABLING SUPPORT FOR PYTHON FEATURES

Enabling Support for Python Features

- There also exist special methods that enable support for certain Python **features**.
- Some examples:
  - Iteration (e.g., in for loops): `__iter__(self)`
  - Context managers (**with** statement): `__enter__(self)` and `__exit__(self, exc_type, exc_value, traceback)`
  - Making objects callable (like function invocations): `__call__(self, ...)`
  - `...`

IMPLEMENTATION

Implementation

- How do we have to implement all these special methods?

Implementation

- How do we have to implement all these special methods?
- **Read the specification**, which lists the requirements and contains the necessary information!

Implementation

- How do we have to implement all these special methods?
- **Read the specification**, which lists the requirements and contains the necessary information!
- Useful links:
  - Data model (among other information, listing the specification of special methods):
    `https://docs.python.org/3/reference/datamodel.html`
  - Python's built-in types:
    `https://docs.python.org/3/library/stdtypes.html`
  - Glossary (listing important terms):
    `https://docs.python.org/3/glossary.html`