Week 10

Advanced Course in Programming

17.11.2023

Last week

NOTE: problem saving Part 9 recording - version from previous year available

Objects in data structures

Objects as parameters

Objects as attributes

Encapsulation

Private members

Static members

Need for specialization

```python
class Student:

    def __init__(self, name: str, id: str, email: str, credits: str):
        self.name = name
        self.id = id
        self.email = email
        self.credits = credits

class Teacher:

    def __init__(self, name: str, email: str, room: str, teaching_years: int):
        self.name = name
        self.email = email
        self.room = room
        self.teaching_years = teaching_years
```

Common superclass

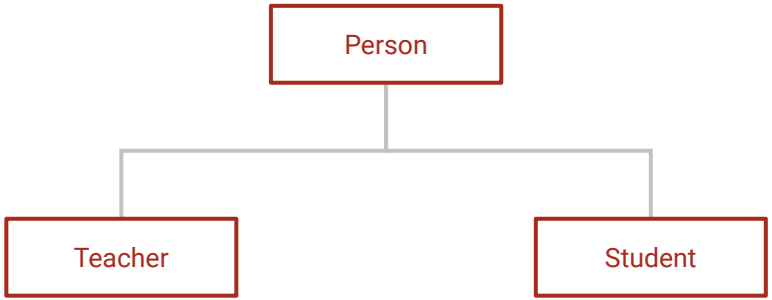
```
class Person:

    def __init__(self, name: str, email: str):
        self.name = name
        self.email = email
```

Terminology

Teacher and Student **inherit** the class Person

Person is the **superclass** of classes Teacher and Student

Teacher and Student are **subclasses** of class Person



Inheriting members

Subclass inherits all members of superclass

Subclass can refer to superclass members *if they are not private*!

```
class Bookshelf(BookContainer):

    def __init__(self):
        super().__init__()
```

Referring superclass members

Subclass can call superclass methods

```
class PlatinumCard(BonusCard):

    def __init__(self):
        super().__init__()

    def calculate_bonus(self):
        # Call the method in the base class
        bonus = super().calculate_bonus()

        # ...and add five percent to the total
        bonus = bonus * 1.05
        return bonus
```

Private members

Private members are also hidden from subclasses

```
def __init__(self):
    self._muistiinpanot = []
```

Sort of solution: protected members

Visibility

Access modifier	Example	Visible to client	Visible to derived class
Public	`self.name`	yes	yes
Protected	`self._name`	no	yes
Private	`self.__name`	no	no

Class type object

Class can return an object of it's own type

```
class Product:
    def __init__(self, name: str, price: float):
        self.__name = name
        self.__price = price

    def __str__(self):
        return f"{self.__name} (price {self.__price})"

    def product_on_sale(self):
        on_sale = Product(self.__name, self.__price * 0.75)
        return on_sale
```

Operator overloading

Overloading operators is a useful trick to enable comparing objects generated from own classes

```
class Product:
    def __init__(self, name: str, price: float):
        self.__name = name
        self.__price = price

    def __str__(self):
        return f"{self.__name} (price {self.__price})"

    @property
    def price(self):
        return self.__price

    def __gt__(self, another_product):
        return self.price > another_product.price
```

Comparison operators

Operator	Traditional meaning	Name of method
`<`	Less than	`__lt__(self, another)`
`>`	Greater than	`__gt__(self, another)`
`==`	Equal to	`__eq__(self, another)`
`!=`	Not equal to	`__ne__(self, another)`
`<=`	Less than or equal to	`__le__(self, another)`
`>=`	Greter than or equal to	`__ge__(self, another)`

Arithmetic operators

Operator	Traditional meaning	Name of method
`+`	Addition	`__add__(self, another)`
`-`	Subtraction	`__sub__(self, another)`
`*`	Multiplication	`__mul__(self, another)`
`/`	Division (floating point result)	`__truediv__(self, another)`
`//`	Division (integer result)	`__floordiv__(self, another)`

Iteraattorit

Joskus oman luokan iterointi for-lauseella olisi kätevää

Esim. kirjahyllyn kirjat, opiskelijarekisterin opiskelijat, muistikirjan muistiinpanot jne.

Methods for iteration

Method__iter__ initializes the iteration

Method__next__ returns the next iterable item (if any)

```
class Book:
    def __init__(self, name: str, author: str, page_count: int):
        self.name = name
        self.author = author
        self.page_count = page_count

class Bookshelf:
    def __init__(self):
        self._books = []

    def add_book(self, book: Book):
        self._books.append(book)

    # This is the iterator initialization method
    # The iteration variable(s) should be initialized here
    def __iter__(self):
        self.n = 0
        # the method returns a reference to the object itself as
        # the iterator is implemented within the same class definition
        return self

    # This method returns the next item within the object
    # If all items have been traversed, the StopIteration event is raised
    def __next__(self):
        if self.n < len(self._books):
            # Select the current item from the list within the object
            book = self._books[self.n]
            # increase the counter (i.e. iteration variable) by one
            self.n += 1
            # return the current item
            return book
        else:
            # All books have been traversed
            raise StopIteration
```

Next week

List comprehension

Dictionary comprehension

Recursion