Python for Beginners

Object Orientation (1)

Classes

```
class Person:

    def say_hi(self):
        print('Hello!')
```

- Use camel case for the class name
- Methods must have the parameter 'self' at least!

Creating and Using Objects

```
# create a new object with the default constructor
p = Person()

# print the standard string representation of an object
print(p)
→ <__main__.Person object at 0x02DE2630>

# call the method say_hi() of the object p
p.say_hi()
→ Hello!
```

Unique Object ID

Insider know-how about the “self parameter”:
The call: `p.say_hi()` is in fact this call: `Person.say_hi(p)`

Object Encapsulation

- All fields and methods are public by default ☹
- Real protection of fields and methods is not possible.
- But fields and methods can be marked by the following naming conventions:
  - Leading underscore: `_fieldname` … this is an internal field that should not be used from outside (just a weak usage indicator)
  - Two leading underscores: `__field_name` … this field will be automatically renamed to `_classname_fieldname` to avoid naming conflicts. So the field is hidden as a private field, but not really)
  - Two leading and trailing underscores: reserved for special usage (= magic names, e.g. `__init__` or `__main__` etc.)

Constructor

```
class Rectangle:
    # constructor with optional named params
    def __init__(self, height=0, width=0):
        # the fields are created inside the constructor at first use
        self.__height = height
        self.__width = width

obj1 = Rectangle()                          # use defaults
obj2 = Rectangle(23)                        # set x
obj3 = Rectangle(3, 4)                      # set x and y
obj4 = Rectangle(width=4, height=3)         # set x and y by name
```

- The magic method __init__() is called automatically when the object is created
- The fields are created in the constructor during initialization
- Constructor overloading is realized by optional parameters

Getter and Setter Methods

```
# getter and setter methods
def get_height(self):
    return self.__height

def set_height(self, value):
    self.__height = value

def get_width(self):
    return self.__width

def set_width(self, value):
    self.__width = value

def get_area(self):
    return self.__height * self.__width
```

- Use lowercase and underscore for the method names

Properties

```
# properties
Height = property(get_height, set_height)
Width = property(get_width, set_width)
Area = property(get_area)  # only with getter
```

- The property function needs the getter as the first param and/or the setter as the second param to create a property

```
# property usage
obj1.Height = 3
```

Hiding Getters and Setters

```
# getter and setter methods
def __get_height(self):
    return self.__height

def __set_height(self, value):
    self.__height = value

def __get_width(self):
    return self.__width

def __set_width(self, value):
    self.__width = value

def __get_area(self):
    return self.__height * self.__width

# properties
Height = property(__get_height, __set_height)
Width = property(__get_width, __set_width)
Area = property(__get_area)   # only with getter
```

- Hide the getters and setter with two leading underscores if you want to offer only the properties

String Representation

```
# the magic to string method
def __str__(self):
    return f'{{height:{self.Height}, width:{self.Width}}}'

# the magic string representation method
def __repr__(self):
    return self.__str__() # just use the __str__ method
```

```
print(obj)  # __str__() is called automatically
→ {height:3, width:4}
```

Comparing Objects

```
# the magic equals method
def __eq__(self, other):
    if isinstance(other, Rectangle):
        return self.Area == other.Area
    else:
        return False



obj1 = Rectangle(3, 4)
obj2 = Rectangle(width=4, height=3)

print(obj1 == obj2) # obj1.__eq__(obj2) is called automatically
→ True
```

Comparing Objects (2)

- Implement the magic methods for all six possible comparisons:

`self <  other`	`__lt__(self, other)`
`self <= other`	`__le__(self, other)`
`self == other`	`__eq__(self, other)`
`self != other`	`__ne__(self, other)`
`self >  other`	`__gt__(self, other)`
`self >= other`	`__ge__(self, other)`

Sort a List of Objects

```
# implement the magic lower than method
def __lt__(self, other):
    if isinstance(other, Rectangle):
        return self.Area < other.Area
    else:
        return False

# create a list of Rectangle objects
rectangle_list = [
    Rectangle(height=5, width=6),
    Rectangle(height=3, width=7),
    Rectangle(height=4, width=3),
]

print(rectangle_list)
→ [{height:5, width:6}, {height:3, width:7}, {height:4, width:3}]

# sort the list in ascending natural order
rectangle_list.sort()

print(rectangle_list)
→ [{height:4, width:3}, {height:3, width:7}, {height:5, width:6}]
```

Sort Custom Order

- Use a lambda expression to perform a custom sort

```
rectangle_list.sort(key = lambda x: x.Width)
```

→Sort every Rectangle object x by x.Width

→`[{height:4, width:3}, {height:5, width:6}, {height:3, width:7}]`

Modules

- A module is a python file that can be imported from another module

main_v1.py: importing the module

```
import my_module

obj = my_module.MyClass()
```

my_module.py

```
class MyClass:
    .
    .
    .
```

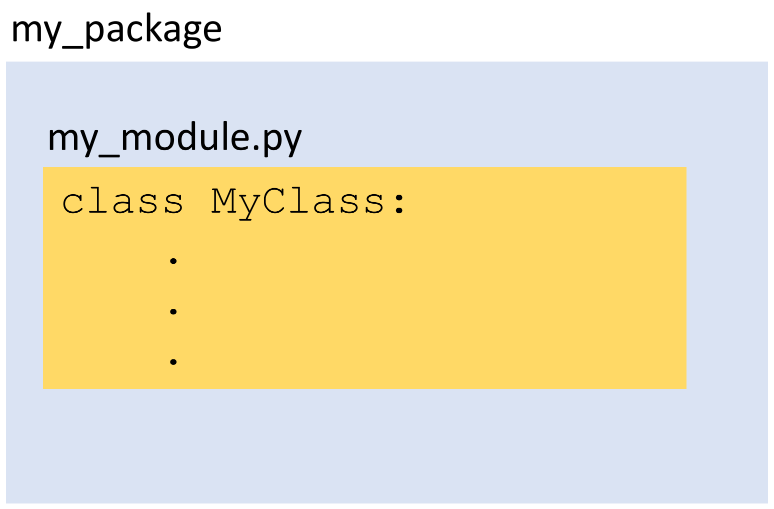
main_v2.py: importing the class of the module

```
from my_module import MyClass

obj = MyClass()
```

Packages

- A package is a folder where modules (= python files) are stored
- You cannot import a package but a module of a package

```python
# import a module of a package
import my_package.my_module
# or
from my_package import my_module

# import a class of a module of a package
from my_package.my_module import MyClass
```



```python
class MyClass:
      .
      .
      .
```