Lab 3A: Introductory Object-oriented Programming in Python

Student: abcde123

Student: abcde123

2. Introduction

Object-oriented Programming

The point of Object-oriented Programming is to support encapsulation and the DRY (Don't Repeat Yourself) principle without things getting out of hand. Often, software architects (those high-level programmers who are responsible for how large systems are designed on a technical level) talk about Object-oriented design or Object-oriented analysis. The point of this is to identify the necessary *objects* in a system. An object in this sense is not exactly the same as a Python object but rather a somewhat higher level logical unit which can reasonably be thought of as an independent component within the system. These high level objects might then be further subdivided into smaller and smaller objects and at a some level the responsibility shifts from the system architect to the team or individual developer working on a specific component. Thus, Object-oriented thinking is necessary for anyone developing code which will be integrated with a larger system, for instance a data scientist implementing analytics tools.

OOP in Python

Python implements the Object-oriented paradigm to a somewhat larger degree than the Functional paradigm. However, there are features considered necessary for *strict* object-oriented programming missing from Python. Mainly, we are talking about data protection. Not in a software security sense, but in the sense of encapsulation. There is no simple way to strictly control access to member variables in Python. This does not affect this lab in any way but is worth remembering if one has worked in a language such as Java previously.

3. Simple instance tests in Python

Note: some of these questions will be extremely simple, and some might prove trickier. Don't expect that the answer needs to be hard.

In [2]:

```
class Person:
    def __init__(self, name):
        self.name = name
        self.age = 0            # Age should be non-negative.

    def get_age(self):
        """Return the Person's age, a non-negative number."""
        return self.age

    def return_five(self):
        """Return 5. Dummy function."""
        return 5

Jackal = Person

president = Person("Jeb")
psec = Jackal("CJ Cregg")
```

a) Change the age of the `president` to 65 (`psec` should be unaffected).

In [2]:

```
president.age = 65
```

[Note: This mode of operation is sometimes considered poor OOP. We will remedy this later.]

b) How many `Person` instances are there? One, two or three?

In [3]:

```
for obj in [Jackal, president, psec]:
    print(type(obj))
Jackal.name
# As it can be seen, 'president' and 'psec' are both objects of class 'Person'. In con
# 'Jackal' is not an object of class 'Person' since the 'name' is not specified for in
```

```
<class 'type'>
<class '__main__.Person'>
<class '__main__.Person'>

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-87a63afadb89> in <module>
      1 for obj in [Jackal, president, psec]:
      2     print(type(obj))
----> 3 Jackal.name
      4 # As it can be seen, 'president' and 'psec' are both objects of clas
s 'Person'. In contrast,
      5 # 'Jackal' is not an object of class 'Person' since the 'name' is no
t specified for initilization.

AttributeError: type object 'Person' has no attribute 'name'
```

c) Consider the following code snippets. What do you think that they will return, and why? Discuss amongst yourselves. After that, run the code and explain the output. You only need to write down your explanation of the output.

In [4]:

```
"Jeb" is Person
```

Out[4]:

```
False
```

In [5]:

```
president is Person
```

Out[5]:

```
False
```

In [ ]:

```
# In both cases, falses are returned since the 'is'-operator only returns true if and
# are exactly the same objects. Since "Jeb" is a string and president is an instance o
# itself, fales are returned in both cases.
```

d) How would you go about checking whether or not the value bound to the name `president` is-a `Person` ?

In [7]:

```
1 type(president) is Person
```

Out[7]:

```
True
```

4. Subclasses

a) Create class `Employee`, a subclass of `Person` with data attributes (fields)

- `__work_days_accrued`
- `__daily_salary`.

These should be *the only* data attributes which you write in your class definition. In particular, you may not duplicate `name` and `age`.

There should be methods

- `work` which increments the numer of work days accrued.
- `expected_payout` which just returns the expected payout for the employee based on the accrued work days and daily salary (but without doing any resets).
- `payout` which returns the accrued salary and resets the number of work days accrued. The `payout` function may not perform any calculations itself.

In [3]:

```
class Employee(Person):
    # Defining constructor.
    def __init__(self, name, work_days = 0, daily_salary = 15):
        Person.__init__(self, name)
        self.__work_days_accured = work_days
        self.__daily_salary = daily_salary
    # Defining class methods.
    def work(self):
        self.__work_days_accured = self.__work_days_accured + 1
    def expected_payout(self):
        return self.__work_days_accured * self.__daily_salary
    def payout(self):
        accured_salary = self.__work_days_accured * self.__daily_salary
        self.__work_days_accured = 0
        return accured_salary

# Ready-made tests.
print("--- Setting up test cases.")
cleaner = Employee(name = "Scruffy")  # Should have daily_salary 15.
josh = Employee(name = "Josh", daily_salary = 1000)
toby = Employee(name = "Toby", daily_salary = 9999)

josh.work()
josh.work()
toby.work()
toby.work()
toby.work()
cleaner.work()

print("--- Testing payout and expected_payout properties.")
assert cleaner.expected_payout() == 15, "default salary should be 15"
assert josh.expected_payout() == 1000*2
assert josh.payout() == 1000*2
assert josh.expected_payout() == 0, "salary should be reset afterwards"
assert toby.payout() == 9999*3, "toby and josh instances should be independent."
print("OK")

print("--- Testing non-data-accessing calls to superclass methods.")
assert josh.return_five() == 5, "Person.return_five should be accessible"
print("OK")

print("--- Testing data that should be set up by initialiser call.")
assert josh.get_age() == 0, "superclass method should be callable, values should not be
josh.age = 9
assert josh.get_age() == 9, "superclass method should be callable"
print("OK")
```

```
--- Setting up test cases.
--- Testing payout and expected_payout properties.
OK
--- Testing non-data-accessing calls to superclass methods.
OK
--- Testing data that should be set up by initialiser call.
OK
```

b) Which public data attributes (fields) does an `Employee` have? Can you access the age of an employee directly (without some transformation of the name)? The daily salary?

In [19]:

```
# An Employee has the two public data attributes 'name' and 'age'.
print(josh.age)

# The other data attributes are all hidden by using '__'. Thus, they are not callable j
print(josh.__daily_salary)
```

```
9

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-19-6d9bf71cda56> in <module>
      3
      4 # The other data attributes are all hidden by using '__'. Thus, they
are not callable from outside.
----> 5 print(josh.__daily_salary)

AttributeError: 'Employee' object has no attribute '__daily_salary'
```

c) Create another subclass of `Person`. This should be called `Student`. Students have the method `work`, which only increases their age by 1/365. Students start out at age 7 (not 0, as persons). You may not modify the `Person` class.

In [4]:

```
class Student(Person):
    def __init__(self, name, age = 7):
        Person.__init__(self, name)
        self.age = age
    def work(self):
        self.age = self.age + 1/365

studious_student = Student(name = "Mike")
assert studious_student.age == 7
```

5. Multiple inheritance

a) Create a subclass `TeachingAssistant`, which so far only contains a constructor. A teaching-assistant is both a Student and an Employee. TA:s daily salaries are always 1.

In [5]:

```
class TeachingAssistant(Employee, Student):
    def __init__(self, name, work_days = 0, daily_salary = 1):
        Employee.__init__(self, name, work_days, daily_salary)
        Student.__init__(self, name)

severus = TeachingAssistant(name = "Severus")
```

b) How would you test if `severus` below is (some kind of) a `Person`? Note that he is (all TA:s are Persons!), and your test should return `True`.

In [6]:

```
severus = TeachingAssistant(name = "Severus")
isinstance(severus, Person)
```

Out[6]:

```
True
```

d) Call the `work` method of a TA object, such as `severus`. What happens? Does their age increase? Their accrued salary? Both? Why is this, in your implementation? [Different groups might have slightly different results here, even if their solutions are correct. Discuss your solution.]

In [7]:

```
print("age before: " + str(severus.age))
print("accrued salary before: " + str(severus.expected_payout()))
severus.work()
print("age afterwards: " + str(severus.age))
print("accrued salary before: " + str(severus.expected_payout()))

# 'severus' is an instance of class TeachingAssistant which inherits of both classes.
# For both classes (Employee and Student), the 'work'-method is defined.
# Since the 'work'-method of TeachingAssistant can only inherit of one of these two cl
# constructor of TeachingAssistant first the parental constructor of Employee, the meth
# the method of Student. Thus, after performing 'work()', age' stays the same while 'ex
```

```
age before: 7
accrued salary before: 0
age afterwards: 7
accrued salary before: 1
```

[Hint: You might want to inspect the `.age` and `.work_days_accrued` attributes of the object. Or simply add a print statement to the work functions that would show you if they were called.]

e) Rewrite the `TeachingAssistant` class definition so that eg `severus.work()` will both increase the age and the expected payout.

In [8]:

```
class TeachingAssistant(Employee, Student):
    def __init__(self, name, work_days = 0, daily_salary = 1):
        Employee.__init__(self, name, work_days, daily_salary)
        Student.__init__(self, name)
        self.__work_days_accured = work_days
        self.__daily_salary = daily_salary
    def work(self):
        self.age = self.age + self.age*1/365
        self.__work_days_accured = self.__work_days_accured + 1
    def expected_payout(self):
        return self.__work_days_accured * self.__daily_salary

severus = TeachingAssistant(name = "Severus")
print("age before: " + str(severus.age))
print("accrued salary before: " + str(severus.expected_payout()))
severus.work()
print("age afterwards: " + str(severus.age))
print("accrued salary before: " + str(severus.expected_payout()))
```

```
age before: 7
accrued salary before: 0
age afterwards: 7.019178082191781
accrued salary before: 1
```

6. Further encapsulation, and properties

a) How would you rewrite the `Person` class so that we can remove `get_age` and provide `.age` as a getter-only property? Use the `@property` syntax. You may rename member attributes.

In [10]:

```
class Person:
    def __init__(self, name):
        self.name = name
        self.__age = 0           # Age should be non-negative.

    def __get_age(self):
        return self.__age

    #def __set_age(self, new_age):
     #   self.__age = new_age

    age = property(fget = __get_age)#, fset = __set_age)

    def return_five(self):
        """Return 5. Dummy function."""
        return 5

president = Person("Jeb")
president.age
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-10-2477b657fa47> in <module>
     18 president = Person("Jeb")
     19 president.age
---> 20 president.age = 3

AttributeError: can't set attribute
```

b) Try to set `president.age` to 100. What happens?

In [11]:

```
president.age = 100
president.age
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-11-02e032065573> in <module>
----> 1 president.age = 100
      2 president.age
      3 # Since we implemented the property-syntax, it works.

AttributeError: can't set attribute
```

c) Now we've modified the `Person` class. What kind of problems do you suspect might come from this when looking at the child classes (without modifying them!)? Give a statement, a sensible line of code, below which demonstrates this.

In [15]:

```
class Student(Person):
    def __init__(self, name, age = 7):
        Person.__init__(self, name)
        self.age = age
    def work(self):
        self.age = self.age + 1/365

studious_student = Student(name = "Mike")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-1701f08ca1b4> in <module>
      6         self.age = self.age + 1/365
      7
----> 8 studious_student = Student(name = "Mike")

<ipython-input-15-1701f08ca1b4> in __init__(self, name, age)
      2     def __init__(self, name, age = 7):
      3         Person.__init__(self, name)
----> 4         self.age = age
      5     def work(self):
      6         self.age = self.age + 1/365

AttributeError: can't set attribute
```

In [ ]:

```
# Provide some test case which fails here.
```

Note: above we changed the public interface of a class, which some other classes or behaviours had come to rely on.

d) Let's say that we previously had the implicit contract "ages are non-negative numbers". This was an idea in the mind of the programmer, but had not implemented in code. Cut-and-paste your modified solution, and add a setter for `age` which enforces this (again, using the decorator `@` syntax). If the age is negative (or something where the comparison fails), a `ValueError` should be raised.

In [ ]:

```
class Person:
    def __init__(self, name):
        self.name = name
        self.__age = 0            # Age should be non-negative.

    def __get_age(self):
        return self.__age

    def __set_age(self, new_age):
        if new_age >= 0:
            self.__age = new_age
        else:
            raise ValueError("age must be non-negative!")

    age = property(fget = __get_age, fset = __set_age)

    def return_five(self):
        """Return 5. Dummy function."""
        return 5

president = Person("Jeb")
president.age
```

Cf the raising of ValueErrors in lab 2A.

e) Given this addition of a somewhat restrictive setter, do the problems with the subclasses that you encountered above disappear? Does this make sense?

In [ ]:

```
# Your answer.
```

General note: If you use Python as a scripting language, having only taken this course, implementing deep or complex structures with multiple inheritances is unlikely to be your first task. What you should recall is that (i) you can do this, (ii) how you can do this technically, (iii) that Python will give you a lot of leeway and (iv) that what you expose in the code matters if someone may later come to rely on it. Especially if the documentation is somewhat lacking, and where contracts are not made explicit in a way that the system will enforce (eg ages should be non-negative). This last part is possibly the most important.

Honourable mentions

This lab by no means treats all useful concepts in OOP or Python. You may want to look up

- Abstract Classes and abc (see the module `abc`, and the more specialised `abc` in the specialised `collections`).
- The concept of "goose typing".
- The concept of mixins. Etc.

Acknowledgments

This lab in 732A74 is by Anders Märak Leffler (2019). The introductory text is by Johan Falkenjack (2018).

Licensed under CC-BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0/).