**Exercises Advanced Programming Topics**

**Exercise 1: OO programming**

In this exercise, you’ll work out an HR payroll system. The example will demonstrate the use of inheritance and how derived classes can provide a concrete implementation of the base class methods.

The HR system needs to process payroll for the company’s employees, but there are 2 different types of employees depending on how their payroll is calculated.

First, create the Employee base class for all employee types. Every Employee must have an id assigned as well as a name. Furthermore, the HR system requires that every Employee processed can call a method .calculate\_payroll(). In the base class, however, this method just returns None.

So your Employee base class looks like this:

class Employee():

    def \_\_init\_\_(self, id, name):

        self.id = id

        self.name = name

    def calculate\_payroll(self):

         pass

Next you create 2 new classes that inherit from the Employee class:

* Firstly add a class called SalaryEmployee. These administrative workers have a fixed salary, so every week they get paid the same amount. So you need to extend the class with the attribute salary and override the calculate\_payroll() method to return this fixed amount.
* Secondly, add an HourlyEmployee class. These class intitializes manufacturing workers that are paid by the hour. So you need to extend this class with the hours\_worked and the hour\_rate required to calculate the payroll. The .calculate\_payroll() method is implemented by returning the hours worked times the hour rate.

Finally, you create a PayrollSystem class that processes payroll. This class contains only one method called calculate\_payroll(self, list\_of\_employees). The method loops over the list with employee objects and prints the corresponding earnings.

Here 4 employee objects are instantiated and added to a list. This list is passed to the calculate\_payroll method.  
  
 #instantiating objects  
anna = SalaryEmployee(1, 'Anna Adams', 1500)  
bert = HourlyEmployee(2, 'Bert Branson', 40, 15)  
chris = SalaryEmployee(3,'Chris Conway',1600)  
dennis = Employee(4,"Dennis")  
employees = [anna,bert, chris, dennis]

hr = PayrollSystem()

hr.calculate\_payroll(employees)

Make sure that this output appears. Depending on the object type passed to the method calculate\_payroll the correct calculation is used and printed.

Text

Description automatically generated

**Exercise 1 extra: Abstract classes in Python**

When a base class Employee object is created, the corresponding salary is None. This is because there is no implementation for the calculate\_payroll() method. The method is declared but contains no code. In OO this method is called an abstract method. A class that contains one or more abstract methods is called an abstract class.

But conform the OO principles you may not be able to instantiate an instance from an abstract class.  
  
In the above sample, however, the object *dennis* is an instance of the Employee class. In fact, Python on its own doesn't provide abstract classes. Yet, Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs). This module is called - for obvious reasons – abc and needs to be imported

# import for exercise 1 extra abstract classes

from abc import ABC, abstractmethod

Then force the base class Employee to inherit from (ABC) and annotate the abstract method.  
Now the employee class looks like this:

# Exercise 1 Extra abstract classes

class Employee(ABC):

    def \_\_init\_\_(self, id, name):

        self.id = id

        self.name = name

    @abstractmethod

    def calculate\_payroll(self):

        pass

When you now try to instantiate an instance of the Employee class, an error is thrown.  


Furthermore, the deriving classes SalaryEmployee and HourlyEmployee must override the calculate\_payroll method.

**Exercise 2**:

Goal is to extract a list containing all the prime numbers in a given interval. For example, all the prime numbers in a range from 1 to 50 returns:

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

There are many solutions for this exercise but here you should use as many built-in functions as possible.

To do that, you can start by coding a function is\_prime\_number() that takes an integer as an argument and returns True if the number is prime and False otherwise.

Then call this function using filter() to extract all the prime numbers in the range between 1 and 50.

**Exercise 3:**

First write 2 functions to convert an incoming temperature from Fahrenheit [°F] to Celsius [°C] and vice versa. These are the formulas you need:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| C = | |  | | --- | | 5 | |  | | 9 | | (F - 32) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| F = | |  | | --- | | 9 | |  | | 5 | | C + 32 |

Secondly, in your main program, initialize 2 lists containing temperatures in °F or °C

ctemps = [0, 12, 34, 100]

ftemps = [32, 65, 100, 212]

Then call the created functions to convert the lists to the other temperature scale using map().

Next, try to perform the conversions on the list data using a Lamba function instead of the defined functions.

Finally, try to use list comprehension to accomplish the same thing.

The output must be 3 times completely identical. Temperatures have been rounded.

[0, 18, 38, 100]

[32, 54, 93, 212]

**Exercise 4**:

Create 2 lists containing the days of the week in English and in Dutch.

weekdays\_english = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]

weekdays\_dutch = ["zon", "ma", "di", "woe", "don", "vrij", "zat"]

Print an overview of the corresponding items:

Table

Description automatically generated

Change the output in:

Text

Description automatically generated

Next add a counter to the output:

Text

Description automatically generated

Finally, create a dictionary containing key value pairs:

{'Sun': 'zon', 'Mon': 'ma', 'Tue': 'di', 'Wed': 'woe', 'Thu': 'don', 'Fri': 'vrij', 'Sat': 'zat'}

**Exercise 5:**

First, create a class City. Each City object consists of a zipcode and a name.

Next, in the main program create 2 lists:

lst1=[2440, 1000,  3500, 2800, 9000]

lst2=["Geel", "Brussel", "Hasselt", "Mechelen", "Gent"]

Finally create a list with City objects and print an overview.

Text

Description automatically generated

**Exercise 6**:

When you’re trying to describe and summarize a sample of data, you probably start by finding its mean, or average. The mean is a quite popular central tendency measurement and is often the first approach to analyzing a dataset.

In some cases, the mean isn’t a good enough central tendency measure for a given sample. Outliers are one of the elements that affect how accurate the mean is. Outliers are data points that differ significantly from other observations in a sample or population.

In this exercise you start with a list of measurements. Two of the measurements (42, 34) however are obviously outliers.

sample = [10, 8, 10, 8, 42, 7, 9, 3, 34, 9, 5, 9, 11,8,7,12]

First calculate the mean of the sample data: -> 12

Next calculate the mean again on a filtered set of the sample data that only contains data points between 5 and 15: -> 8.692307692307692

Try to write the second calculation in one line of Python code using a Lambda function.

Use the mean() out of the basic Python statistics library.

**Extra exercise 6:**

Later in this course, you will learn that in normally distributed samples, outliers are often defined as data points that lie more than two standard deviations from the sample mean.   
  
  
Chart, histogram

Description automatically generated  
The calculation of the standard deviation can be made with the function stdev() which can be called from the statistics library.

Now use this approach to remove the outliers. Now data point ‘3’ is removed as well.  
The mean is now -> 8.285714285714286.