Python Course Cheat Sheet 8

The Object Super Class

```
class MyClass:
     pass
a=MyClass()
dir(a)
```

- dir() returns a list of all the members in the specified object
- We have not declared any members in MyClass, so where is the list coming from?

```
# Test object class
test_object= object()
dir(test_object)
```

This is because every class you create in Python implicitly derives from object

Creating Class Hierarchies

- Inheritance is the mechanism you'll use to create hierarchies of related classes
- Modeling an HR system. The example will demonstrate the use of inheritance and how derived classes can provide a concrete implementation of the base class interface
- Requirment1 : Need to process payroll for the company's employees
- Q is: there are different types of employees depending on how their payroll is calculated

You start by implementing a PayrollSystem class that processes payroll:

```
# Define payroll class
class PayrollSystem:
    def calculate_payroll(self, employees):
        print('Calculating Payroll')
        print('==================')
        for employee in employees:
            print(f'Payroll for: {employee.id} - {employee.name}')
            print(f'- Check amount: {employee.calculate_payroll()}')
            print('')
```

- The PayrollSystem implements a .calculate_payroll()
- calculate_payroll() method that takes a collection of employees and prints their id, name
- check amount using the .calculate_payroll() method exposed on each employee object

Now, you implement a base class Employee that handles the common interface for every employee type:

```
# Define a general calss for employee
class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name
```

- Employee is the base class for all employee types
- Employee must have an id assigned as well as a name
  Now,
- HR system requires that every Employee processed must provide a .calculate_payroll()
- The implementation of that interface differs depending on the type of Employee

Type1: Administrative workers have a fixed salary

```
class SalaryEmployee(Employee):
    def __init__(self, id, name, weekly_salary):
        super().__init__(id, name)

        self.weekly_salary = weekly_salary

    def calculate_payroll(self):

        return self.weekly_salary
```

- Derived class SalaryEmployee that inherits Employee
- Class is initialized with the id and name required by the base class, and you use super()
- Addition: SalaryEmployee also requires a weekly_salary initialization parameter that represents the amount the employee makes per week
- class provides the required .calculate_payroll() method used by the HR system

Type2: The company also employs manufacturing workers that are paid by the hour

```
class HourlyEmployee(Employee):
    # calss attribure gu_rate=10
    def __init__(self, id, name, hours_worked, hour_rate):
        super().__init__(id, name)

        self.hours_worked = hours_worked
        self.hour_rate = hour_rate

    def calculate_payroll(self):
        return self.hours_worked * self.hour_rate
```

- Initialized with id and name, like the base class
- hours_worked and the hour_rate required to calculate the payroll
- calculate_payroll() method is implemented by returning the hours worked times the hour rate

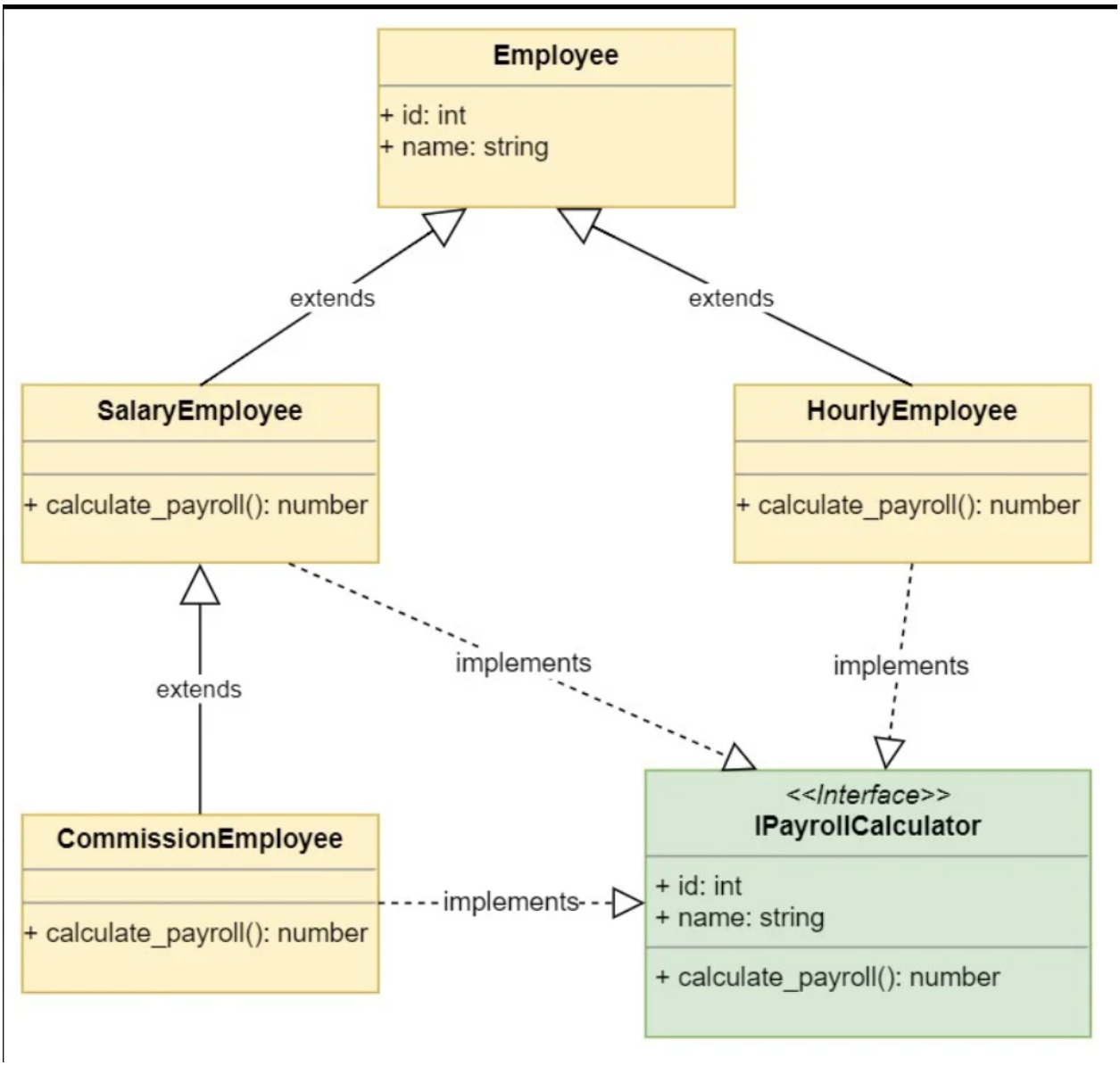
Type3: Company employs sales associates that are paid through a fixed salary plus a commission

```
class CommissionEmployee(SalaryEmployee):
    def __init__(self, id, name, weekly_salary, commission):
#         super().__init__(id, name)
#         self.weekly_salary=weekly_salary
#         self.commission=commission
        super().__init__(id, name, weekly_salary)
        self.commission = commission

    def calculate_payroll(self):
#         weekly_salary=weekly_salary-(weekly_salary*.10)
#         return weekly_salary+self.commesion

        fixed = super().calculate_payroll()
        return fixed + self.commission
```

- CommissionEmployee from SalaryEmployee because both classes have a weekly_salary to consider.
- CommissionEmployee is initialized with a commission value that is based on the sales
- calculate_payroll() leverages the implementation of the base class to retrieve the fixed salary and adds the commission value
- CommissionEmployee derives from SalaryEmployee, you have access to the weekly_salary property directly
- Problem with accessing the property directly is that if the implementation of SalaryEmployee.calculate_payroll() changes, you'll have to also change the implementation of CommissionEmployee.calculate_payroll()



- The derived classes implement the IPayrollCalculator interface
- PayrollSystem.calculate_payroll() implementation requires that the employee objects passed contain an id, name, and calculate_payroll
- Interfaces are represented similarly to classes with the word interface above the interface name
  The application creates its employees and passes them to the payroll system to process payroll:

```
salary_employee = SalaryEmployee(1, 'emp1', 1500)
hourly_employee = HourlyEmployee(2, 'emp2', 40, 15)
commission_employee = CommissionEmployee(3, 'emp3', 1000, 250)
payroll_system = PayrollSystem()
payroll_system.calculate_payroll([
    salary_employee,
    hourly_employee,
    commission_employee
])
```

- creates three employee objects, one for each of the derived classes
- creates the payroll system and passes a list of the employees to its .calculate_payroll() method, which calculates the payroll for each employee
- Employee base class doesn’t define a .calculate_payroll() method

```
employee = Employee(1, 'Invalid')
payroll_system = PayrollSystem()
payroll_system.calculate_payroll([employee])
```

- It can’t .calculate_payroll() for an Employee
- To meet the requirements of PayrollSystem, you’ll want to convert the Employee class, which is currently a concrete class, to an abstract class
- No employee is ever just an Employee, but one that implements .calculate_payroll()

Abstract Base Classes in Python

- Employee class in the example above is what is called an abstract base class.
- Abstract base classes exist to be inherited, but never instantiated.
- Python provides the abc module to define abstract base classes
  The abc module in the Python standard library provides functionality to prevent creating objects from abstract base classes.

```
from abc import ABC, abstractmethod

# this is abstarct class
class Employee(ABC):
    def __init__(self, id, name):
        self.id = id
        self.name = name

    def get_id():

        return self.id
    def get_name():

        return self.name

    @abstractmethod
    def calculate_payroll(self):
        pass
```

- You derive Employee from ABC, making it an abstract base class. Then, you decorate the .calculate_payroll() method with the @abstractmethod decorator. This change has two nice side-effects:
- Telling users of the module that objects of type Employee can’t be created.
- Telling other developers working on the hr module that if they derive from Employee, then they must override the .calculate_payroll() abstract method

```
employee = Employee(1, 'abstract')
```

Implementation Inheritance vs Interface Inheritance

When you derive one class from another, the derived class inherits both:

- The base class interface: The derived class inherits all the methods, properties, and attributes of the base class.
- The base class implementation: The derived class inherits the code that implements the class interface.

```
class AnonymousEmployee():
    def __init__(self, id, name):
        self.id = id
        self.name = name

    def calculate_payroll(self):
        return 1000000
```

```
salary_employee = SalaryEmployee(1, 'emp1', 1500)
hourly_employee = HourlyEmployee(2, 'emp2', 40, 15)
commission_employee = CommissionEmployee(3, 'emp3', 1000, 250)
anonymous_employee = AnonymousEmployee(20000, 'Anonymous')
payroll_system = PayrollSystem()
payroll_system.calculate_payroll([
    salary_employee,
    hourly_employee,
    commission_employee,
    anonymous_employee
])
```

- AnonymousEmployee class doesn’t derive from Employee, but it exposes the same interface required by the PayrollSystem
- PayrollSystem.calculate_payroll() requires a list of objects that implement the following interface:
- An id property or attribute that returns the employee’s id
- A name property or attribute that represents the employee’s name
- A .calculate_payroll() method that doesn’t take any parameters and returns the payroll amount to process
- All these requirements are met by the DisgruntledEmployee class, so the PayrollSystem can still calculate its payroll.
- In Python, you don’t have to explicitly declare an interface. Any object that implements the desired interface can be used in place of another object. This is known as duck typing. Duck typing is usually explained as “if it behaves like a duck, then it’s a duck.”

Why you should use inheritance instead of just implementing the desired interface?

- Use inheritance to reuse an implementation
- Implement an interface to be reused

Let’s clean up our example:

```
class PayrollSystem:
    def calculate_payroll(self, employees):
        print('Calculating Payroll')
        print('==================')
        for employee in employees:
            print(f'Payroll for: {employee.id} - {employee.name}')
            print(f'- Check amount: {employee.calculate_payroll()}')
            print('')

class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name

class SalaryEmployee(Employee):
    def __init__(self, id, name, weekly_salary):
        super().__init__(id, name)
        self.weekly_salary = weekly_salary

    def calculate_payroll(self):
        return self.weekly_salary

class HourlyEmployee(Employee):
    def __init__(self, id, name, hours_worked, hour_rate):
        super().__init__(id, name)
        self.hours_worked = hours_worked
        self.hour_rate = hour_rate

    def calculate_payroll(self):
        return self.hours_worked * self.hour_rate

class CommissionEmployee(SalaryEmployee):
    def __init__(self, id, name, weekly_salary, commission):
        super().__init__(id, name, weekly_salary)
        self.commission = commission

    def calculate_payroll(self):
        fixed = super().calculate_payroll()
        return fixed + self.commission
```

- removed the import of the abc module since the Employee class doesn’t need to be abstract.
- removed the abstract calculate_payroll() method from it since it doesn’t provide any implementation
- inheriting the implementation of the id and name attributes of the Employee class
- calculate_payroll() is just an interface to the PayrollSystem.calculate_payroll() method, you don’t need to implement it in the Employee base class

The Class Explosion Problem

- inheritance can lead you to a huge hierarchical structure of classes that is hard to understand and maintain