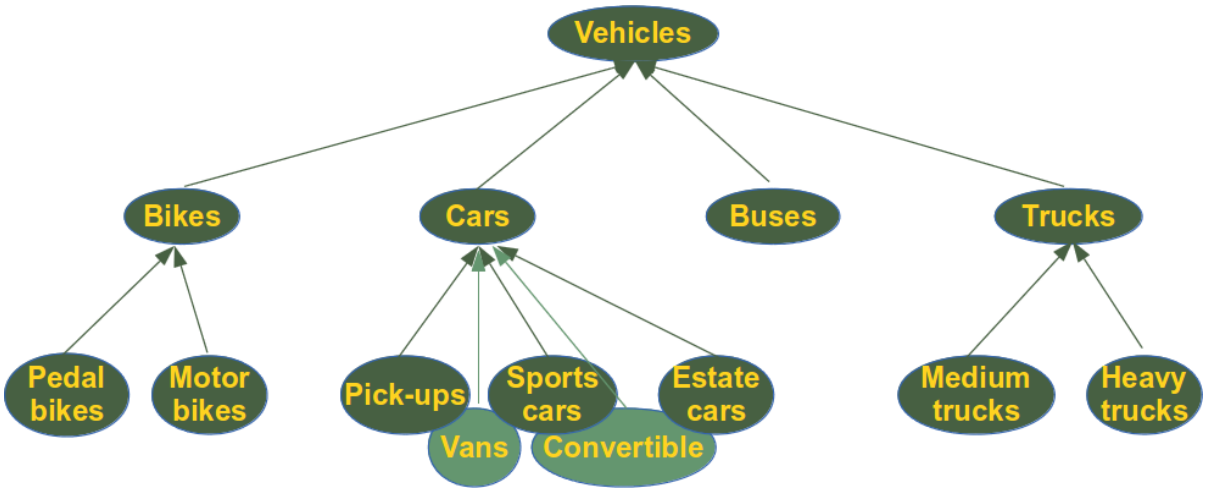
INHERITANCE

INTRODUCTION



Every object-oriented programming language would not be worthy to look at or use, if it weren't to support inheritance. Of course, Python supports inheritance, it even supports multiple inheritance. Classes can inherit from other classes. A class can inherit attributes and behaviour methods from another class, called the superclass. A class which inherits from a superclass is called a subclass, also called heir class or child class. Superclasses are sometimes called ancestors as well. There exists a hierarchy relationship between classes. It's similar to relationships or categorizations that we know from real life. Think about vehicles for example. Bikes, cars, buses and trucks are vehicles. pick-ups, vans, sports cars, convertibles and estate cars are all cars and by being cars they are vehicles as well. We could implement a vehicle class in Python, which might have methods like accelerate and brake. Cars, Buses and Trucks and Bikes can be implemented as subclasses which will inherit these methods from vehicle.

SYNTAX AND SIMPLE INHERITANCE EXAMPLE

We demonstrate inheritance in a very simple example. We create a Person class with the two attributes "firstname" and "lastname". This class has only one method, the Name method, essentially a getter, but we don't have an attribute name. This method is a further example for a "getter", which creates an output by creating it from more than one private attribute. Name returns the concatenation of the first name and the last name of a person, separated by a space. It goes without saying that a useful person class would have additional attributes and further methods.

This chapter of our tutorial is about inheritance, so we need a class, which inherits from Person. So far employees are Persons in companies, even though they may not be treated as such in some firms. If we created an Employee class without inheriting from Person, we would have to define all the attributes

and methods in the Employee class again. This means we would create a design and maybe even a data redundancy. With this in mind, we have to let Employee inherit from Person.

The syntax for a subclass definition looks like this:

```
class DerivedClassName(BaseClassName):
    pass
```

Of course, usually we will have an indented block with the class attributes and methods instead of merely a pass statement. The name BaseClassName must be defined in a scope containing the derived class definition. With all this said, we can implement our Person and Employee class:

```
class Person:

    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def Name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):

    def __init__(self, first, last, staffnum):
        Person.__init__(self,first, last)
        self.staffnumber = staffnum

    def GetEmployee(self):
        return self.Name() + ", " +  self.staffnumber

x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")

print(x.Name())
print(y.GetEmployee())
```

Our program returns the following output:

```
$ python3 person.py
Marge Simpson
Homer Simpson, 1007
```

The __init__ method of our Employee class explicitly invokes the __init__method of the Person class. We could have used super instead. super().__init__(first, last) is automatically replaced by a call to the superclasses method, in this case __init__:

```
    def __init__(self, first, last, staffnum):
        super().__init__(first, last)
        self.staffnumber = staffnum
```

Please note that we used super()) without arguments. This is only possible in Python3. We could have written "super(Employee, self).__init__(first, last, age)" which still works in Python3 and is compatible with Python2.

OVERLOADING AND OVERRIDING

Instead of using the methods "Name" and "GetEmployee" in our previous example, it might have been better to put this functionality into the "__str__" method. In doing so, we gain a lot, especially a leaner design. We have a string casting for our classes and we can simply print out instances. Let's start with a __str__ method in Person:

```
class Person:

    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def __str__(self):
        return self.firstname + " " + self.lastname

class Employee(Person):

    def __init__(self, first, last, staffnum):
        super().__init__(first, last)
        self.staffnumber = staffnum


x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")

print(x)
print(y)
```

The output looks like this:

```
$ python3 person2.py
Marge Simpson
Homer Simpson
```

First of all, we can see that if we print an instance of the Employee class, the __str__ method of Person is used. This is due to inheritance. The only problem we have now is the fact that the output of "print(y)" is not the same as the "print(y.GetEmployee())". This means that our Employee class needs its own __str__ method. We could write it like this:

```
    def __str__(self):
        return self.firstname + " " + self.lastname + ", " +
self.staffnumber
```

But it is a lot better to use the __str__ method of Person inside of the new definition. This way, we can make sure that the output of the Employee __str__method will automatically change, if the __str__ method from the superclass Person changes. We could for example add a new attribute age in Person:

```
class Person:

    def __init__(self, first, last, age):
        self.firstname = first
        self.lastname = last
        self.age = age

    def __str__(self):
        return self.firstname + " " + self.lastname + ", " +
str(self.age)

class Employee(Person):

    def __init__(self, first, last, age, staffnum):
        super().__init__(first, last, age)
        self.staffnumber = staffnum

    def __str__(self):
        return super().__str__() + ", " +  self.staffnumber


x = Person("Marge", "Simpson", 36)
y = Employee("Homer", "Simpson", 28, "1007")

print(x)
print(y)
```

We have overridden the method __str__ from Person in Employee. By the way, we have overridden __init__ also. Method overriding is an object-oriented programming feature that allows a subclass to provide a different implementation of a method that is already defined by its superclass or by one of its superclasses. The implementation in the subclass overrides the implementation of the superclass by providing a method with the same name, same parameters or signature, and same return type as the method of the parent class.

Overwriting is not a different concept but usually a term wrongly used for overriding!

In the context of object-oriented programming, you might have heard about "overloading" as well. Overloading is the ability to define the same method, with the same name but with a different number of arguments and types. It's the ability of one function to perform different tasks, depending on the

number of parameters or the types of the parameters.

Let's look first at the case, in which we have the same number of parameters but different types for the parameters. When we define a function in Python, we don't have to and we can't declare the types of the parameters. So if we define the function "successor" in the following example, we implicitly define a family of function, i.e. a function, which can work on integer values, one which can cope with float values and so. Of course, there are types which will lead to an error if used:

```
>>> def successor(number):
...     return number + 1
...
>>> successor(1)
2
>>> successor(1.6)
2.6
>>> successor([3,5,9])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in successor
TypeError: can only concatenate list (not "int") to list
>>>
```

You can skip the following paragraphs with the comparisons with C++, if you want to.

This course is not about C++ and we have so far avoided using any C++ code. We want to make an exception now, so that you can see, how overloading works in C++. While we had just one definition in Python, we have two function definitions in C++, i.e. one for the type "int" and one for "double":

```
#include <iostream>
#include <cstdlib>

using namespace std;

int successor(int number) {
    return number + 1;
}

double successor(double number) {
    return number + 1;
}

int main() {

    cout << successor(10) << endl;
```

```
    cout << successor(10.3) << endl;

    return 0;
}
```

Having a function with a different number of parameters is another way of function overloading. The following C++ program shows such an example. The function f can be called with either one or two integer arguments:

```
#include <iostream>
using namespace std;


int f(int n);
int f(int n, int m);

int main() {

    cout << "f(3): " << f(3) << endl;
    cout << "f(3, 4): " << f(3, 4) << endl;
    return 0;
}

int f(int n) {
    return n + 42;
}
int f(int n, int m) {
    return n + m + 42;
}
```

This doesn't work in Python, as we can see in the following example. The second definition of f with two parameters redefines or overrides the first definition with one argument. Overriding means that the first definition is not available anymore. This explains the error message:

```
>>> def f(n):
...     return n + 42
...
>>> def f(n,m):
...     return n + m + 42
...
>>> f(3,4)
49
```

```
>>> f(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes exactly 2 arguments (1 given)
>>>
```

If we need such a behaviour, we can simulate it with default parameters:

```
def f(n, m=None):
    if m:
        return n + m +42
    else:
        return n + 42
```

The * operator can be used as a more general approach for a family of functions with 1, 2, 3, or even more parameters:

```
def f(*x):
    if len(x) == 1:
        return x[0] + 42
    else:
        return x[0] + x[1] + 42
```