Introduction to Python Programming

Introduction to Object-Oriented Programming

Annemarie Friedrich (anne@cis.uni-muenchen.de)
Centrum für Informations- und Sprachverarbeitung LMU München

Software objects represent real-life objects

Object-Oriented Programming (OOP) is a programming paradigm in which the basic building block is the *software object* (often just called *object*). The main idea is to represent real-life objects as software objects, which combine characteristics (*attributes*) and behaviors (*methods*). For instance, a bank account is a real-life object. Assume we want to program an application for a bank. Each account is represented by an account object in our program. If our bank has two customers, say Anne and Stefan, we will use two account objects, `annesAcc` and `stefansAcc`.

Attributes represent data

On the one hand, these two objects will differ in some respects. For instance, Stefan's account balance is $1000 while Anne's account balance is only $200. Also, the objects will have different account IDs and account holders. The account balance, account number (=ID) and account holder are specific to each account and describe the *state* of an account. We also call the balance, number and holder *attributes* of the account object. Attributes are used to represent the *data* associated with a real-life object. Some of the attributes may change over time, for instance the account balance can go up or down depending on whether Anne and Stefan deposit or withdraw money from their accounts.

`Object` **Attributes**	`annesAccount`	`stefansAccount`
number	1	2
holder	'Anne'	'Stefan'
balance	200	1000

Classes are blueprints for objects

Objects are created (*instantiated*) from a definition called *class* - programming code that can define attributes and methods. Classes are like blueprints, they are a design for objects. By convention, class names should start with a capital letter. In order to create our account objects, we define an `Account` class. The `class` keyword tells Python that we are starting to define a class. So far, our class doesn't contain anything - `pass` is a special Python keyword that stands for 'nothing happens here'.

However, we can use this class to create account objects as shown in lines 8 & 9 of the main part of the program. We create new objects of this class by 'calling' the class. The dot notation can be used to assign or access (read) attributes as shown in the example. However, this is not the best way to deal with attributes - we will hear more about this later.

```python
 1  class Account:
 2      ''' a class for objects representing an account '''
 3      pass
 4
 5  # Main part of the program
 6  if __name__ == "__main__":
 7      # Creating objects
 8      annesAcc = Account()
 9      stefansAcc = Account()
10      # Assigning attributes
11      annesAcc.number = 1
12      annesAcc.holder = "Anne"
13      annesAcc.balance = 200
14      stefansAcc.number = 2
15      stefansAcc.holder = "Stefan"
16      stefansAcc.balance = 1000
17      # Accessing (reading) attributes
18      print("Balance Anne:", annesAcc.balance)
19      print("Balance Stefan:", stefansAcc.balance)
```

Methods implement behavior

On the other hand, the two account objects have some commonalities, for instance we can withdraw money from either account or deposit money into either account. In OOP, such functionalities associated with objects are called *behaviors* and are programmed as *methods*. Methods are essentially functions that belong to a class. We do not have to write the `withdraw(amount)` and `deposit(amount)` methods separately for the `annesAcc` and the `stefansAcc` objects. Instead, we write the methods that are shared by all objects of a class once in the class definition, and the methods will be available for all objects that are created by calling this class. This is one reason why classes are so terribly useful.

We write an `Account` class that is a design for account objects, and represent it using a *UML class diagram*. Unified Modeling Language (UML) is a visualization standard used in object-oriented engineering.

Name of the class
Attributes
Methods

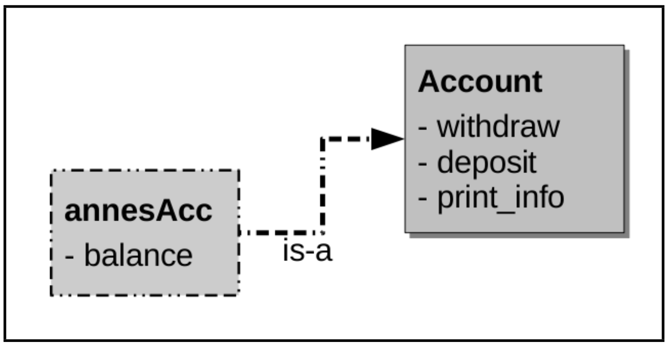
Account
id holder balance

deposit(amount)
withdraw(amount)

```python
class Account(object):
    ''' a class for objects representing an account '''
    # METHODS
    def withdraw(self, amount):
        self.balance -= amount
    def deposit(self, amount):
        self.balance += amount
    def print_info(self):
        print("Balance:", self.balance)

if __name__ == "__main__":
    annesAcc = Account()
    annesAcc.balance = 200
    annesAcc.deposit(500)
    annesAcc.withdraw(20)
    annesAcc.print_info()
```

The methods `deposit(self, amount)`, `withdraw(self, amount)` and `print_info(self)` are *instance methods* of the `Account` class. This means that they are designed to operate on objects that have been created from this class. Technically, instance methods are required to have a first parameter which is called `self` (it could be called something else, too, but `self` is a very strong convention in Python).

The following image shows that we have an object, `annesAcc`, which is linked to the class `Account`. The `balance` attribute is only available at the particular object, but the methods that have been defined in the class are available in the class.



We can call the class's methods 'on an object' using the dot notation: `annesAcc.deposit(500)`. In the code written for the class, however, the method looks like this: `deposit(self, amount)`. It has two parameters, but we called it only with one (`amount`). So why is that?

The following happens here: Python knows that `annesAcc` is an object of the `Account` class (because we created the object by calling the class). Python executes the `deposit(self, amount)` method of the `Account` class and assigns the object on which the method was called to the `self` parameter. This means that the following two lines of code are equivalent:

```python
annesAccount.deposit(500)
Account.deposit(annesAccount, 500)
```

When operating on instance objects, you should always use the first way, because it leaves the job to look up the class to Python and your code will be more flexible. This will become clear later when we talk about inheritance.

Constructors are useful for initialization

The code for the class as written above is not very robust. It does not ensure that the account object actually has a `balance` attribute before trying to print it out. If we just add the following two lines of code to the main program, Python gives us an error, complaining about the fact that we try to print out a `balance` attribute of the `stefansAcc` object, although we never created a balance attribute for this object.

```
stefansAcc = Account()
stefansAcc.print_info()
```

Remember to always make sure that any attribute that your methods try to access actually exists. In this case, using a *constructor method* is helpful. In Python, a constructor method is in fact an *initialization method*.

```
1  class Account(object):
2      ''' a class representing an account '''
3      # CONSTRUCTOR
4      def __init__(self, num, person):
5          self.balance = 0
6          self.number = num
7          self.holder = person
8      # METHODS
9      ...
10
11 # Main part of the program
12 # Execution starts here!
13 if __name__ == "__main__":
14     annesAcc = Account(1, "Anne")
15     annesAcc.deposit(200)
16     annesAcc.print_info()
17     stefansAcc = Account(2, "Stefan")
18     stefansAcc.deposit(1000)
19     stefansAcc.print_info()
```

In Python, constructor methods must have the name `__init__` (note the two underscores at either side), and it must have a `self` parameter.

When creating an object, such as in line 183, the following happens: Python creates a new object, `annesAcc` and then calls the constructor method `__init__(self, num, person)`. The values of the parameters `num` and `person` are given when creating the object by calling the class: `Account(1, "Anne")`. Again, the new `annesAcc` object is automatically assigned to the `self` parameter, and inside the constructor method, we can now assign the necessary attributes to the object.

The Python statement `annesAcc = Account(1, "Anne")` triggers the following steps:

1. A new object is created from the `Account` class and assigned to the variable `annesAcc`.
2. The constructor method of the `Account` class is called. In this step, the newly created object is assigned to the `self` parameter of the constructor method. So technically, Python executes `Account.__init__(annesAcc, 1, "Anne")`.
3. Inside the constructor method, the new object is *initialized* (the attributes are set to the given or default values).

In Python, constructor methods do not *create* an object. You can imagine that the creation of an object happens 'under the hood' and the constructor method then just initializes the attributes of the object. (But this is rather a technical detail - just remember that the `__init__(self)` method is executed right after an object has been created.)

Class Design

Generally, when designing a class, you have to ask yourself the following questions:

1. How can I describe the state of my object? This will result in some *attributes*.
2. What do I know about the object before/when creating it? This goes into the *constructor method*.
3. What operations that change the object's attributes will be performed on the object? These operations need to be implemented as *instance methods*.

When designing the `Account` class, the answers to the above questions are as follows:

1. The state of an account is described by its account number, account holder and the current balance.
2. When creating an account, I know the new account number and the holder's name. The default balance of an account is 0. We pass the account number and the holder's name to the constructor method, and set the default balance to 0.
3. We will change the balance attribute of the account when withdrawing or depositing money. Also, a method that prints out all information we have about the state of the account is useful.

Manipulate attributes only via instance methods

It was said earlier that assigning a value to an object's attribute like this is bad style: `stefansAcc.balance = 1000`.

In OOP, an important principle is the one of *data encapsulation*, which means that the attributes of an object should be 'hidden' from manipulations from 'outside' (i.e. from the code that uses the object). The attributes of an object should only be modified using code that was written *within* the class definition. This ensures that the state of the object is always valid. Imagine, Stefan's account balance is $1000, and he wants to withdraw $1500. However, Stefan's account has the restriction that it may not have a negative balance. Assume the (human) teller at the bank forgets about this restriction, sets the balance manually to -$500 and hands the cash to Stefan. You can imagine that this would make the branch manager of the bank quite unhappy when he realizes that Stefan's account is in a state that is not allowed. The `withdraw(self, amount)` method modifies the balance attribute, but we can control this manipulation in the method. For instance, we could prohibit withdrawals that would result in a negative balance.

```python
class Account(object):
    ''' a class representing an account '''
        ...
    # METHODS
    def withdraw(self, amount):
        if amount > self.balance:
            amount = self.balance
        self.balance -= amount
        return amount
   ...

# Main part of the program
if __name__ == "__main__":
    annesAcc = Account(1, "Anne")
    annesAcc.deposit(200)
    print("Trying to withdraw 40:")
    cash1 = annesAcc.withdraw(40)
    print("Yeah, I got:", cash1)
    print("Trying to withdraw 190:")
    cash2 = annesAcc.withdraw(190)
    print("Oh no, I only got:", cash2)
```

Provide for changing attributes by defining setter methods

Sometimes, an attribute has to be changed completely. Assume, for tax reasons, it is preferable for Stefan to change the account holder to its wife. Because we said that assigning a value to an attribute from the outside like `stefansAcc.holder = "Andrea"` is bad style, we provide a *setter* method for this case.

```python
class Account(object):
    ''' a class representing an account '''
   ...
    def set_holder(self, person):
        self.holder = person

# Main part of the program
# Execution starts here!
if __name__ == "__main__":
    stefansAcc = Account(2, "Stefan")
    stefansAcc.deposit(1000)
    stefansAcc.set_holder("Andrea")
    stefansAcc.print_info()
```

In the above code, the setter method simply sets the object's attribute to the given value. This may seem superfluous at the moment, and there are actually other (safer) ways to do this in Python (keyword: properties). However, for now, we go along with the Python programming principle of trusting the code that uses your classes and stick to the following rules in order not to violate the data encapsulation paradigm:

1. Assign values to attributes only via instance methods (setters) or the constructor.
2. Modify the values of attributes only via instance methods.
3. Accessing (reading) the value of attributes like `print(stefansAcc.balance)` is okay.

So what good is that? For instance, inside the setter methods, we can validate the new value, or raise an exception if the new value is not valid. The new value for the account holder attribute has to be a name - or at least a non-empty string that contains only letters and spaces. We could hence write the setter method like this:

```python
def set_holder(self, person):
    if (not type(person) == str):
        raise TypeError
    if not re.match("\W+( \W+)*", person.strip()):
        raise ValueError
    self.holder = person
```

String representations of objects

Often, it is useful to have a meaningful string representation of an object. If we tell Python to print an object, for instance `print(annesAcc)`, Python gives the cryptic answer "<__main__.Account object at 0xb74faf4c>". Instead, we would rather have a string representation that really tells us what's going on with the attributes of the object, such as the information that the `print_info(self)` method gives us. In Python, we can simply add an instance method that has the name `__str__(self)` (no parameters besides `self`!) to the class. This method must return some string that describes the object. We can see such a method in the code listing on the following page.

In some cases, Python automatically realizes that we want a string representation of an object, for instance when calling `print(annesAcc)` or `str(annesAcc)`, it automatically calls the `__str__(self)` method of this object.

When calling `print(annesAcc)`, it prints out whatever the `__str__(self)` method of the `annesAcc` object returns; `str(annesAcc)` returns the string that is returned by the method call `annesAcc.__str__()`.

```python
class Account:
    # METHODS
    def __str__(self):
        res = "*** Account Info ***\n"
        res += "Account ID:" + str(self.number) + "\n"
        res += "Holder:" + self.holder + "\n"
        res += "Balance: " + str(self.balance) + "\n"
        return res

if __name__ == "__main__":
    annesAcc = Account(1, "Anne")
    annesAcc.deposit(200)
    print(annesAcc)
```

Hooks are special functions of a class that are called automatically by Python in some circumstances. The `__str__(self)` method is only one example for this. There is another hook method called `__repr__(self)` which is also supposed to return a string that describes the object. `__str__(self)` is supposed to return a user-friendly description of the object, while `__repr__(self)` should return a description that is useful for developers. Note that `print(anObject)` returns the string returned by the `__str__(self)` method, while printing an object by just typing it in the interactive mode returns the string returned by `__repr__(self)`. Another hook function that we have seen already is the constructor method (`__init__(self, args)`), which is called when creating a new object by calling a class. There are also hook methods that are triggered when operators are applied, for example the operator + triggers the `__add__` method of an object. Note that the names of hook methods usually start and end with two underscores. We will learn more about hooks later.

Classes are objects, too

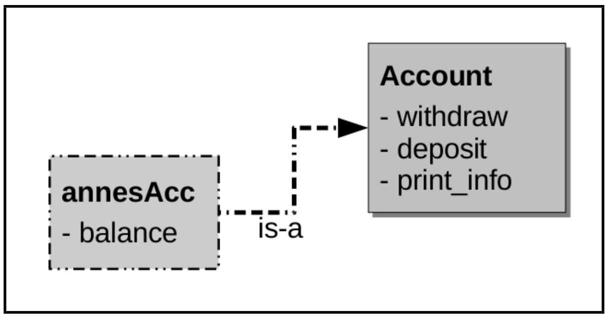
In Python, classes are objects, too. They are created when defining a class using the `class` statement. After defining the `Account` class, exactly one *class object* for the `Account` class becomes available. Each time we call this class (e.g. `annesAcc = Account()`), we create a new *instance object* of this class. The object `annesAcc` is automatically linked to the `Account` class object; we say `annesAcc` 'is-a' `Account`, because it's an instance of the `Account` class.

When we try to access a component (an attribute or a method) of an object using the dot notation in the way `objectName.componentName`, Python first looks in the object itself. Each object can be seen as its own namespace. If Python finds the `componentName` there, it is happy. If it doesn't, it looks in the class object of the class from which the object was created. If it doesn't find the `componentName` there, an error occurs.

Let's look at an example. Assume that the `Account` class has three methods, `withdraw(self, amount)`,`deposit(self, amount)` and `print_info(self)`. We created the object `annesAcc` by calling the `Account` class, and then we assigned an attribute to the `annesAcc` object by writing `annesAcc.balance = 200`. The `annesAcc` object is linked to the class it was created from, as shown in the image on the next page.

When we use the dot notation in order to access an attribute or a method of the `annesAcc` object, Python starts searching for this attribute or component at the object itself. For example, when accessing the balance attribute (`print(annesAcc.balance)`), it finds the attribute right at the object and is happy. Theoretically, you can also define methods only for particular instances. Practically, methods are always defined within classes.

We can call the methods 'on an object' using the dot notation again, like for instance in line 17. Here, Python first looks at the object, but it doesn't have a method called `deposit(amount)`. Python then checks the class from which the object was created, and finds the method there. The `deposit(self, amount)` method of the `Account` class object is then called in a particular way, which we will consider using our example.



Although the method `deposit(self, amount)` has two parameters, we only have to call it with one parameter (the amount of money we intend to deposit). We have already learned thatt he three methods are so-called *instance methods*, which means they should only be called 'on an object', i.e. in the way:
`someObject.method(parameters...)`.

When we define an instance method (inside a class), the first parameter has to be called `self` by convention. When we call the instance method, Python automatically assigns the object on which the method was called to this `self` parameter. So this is what happens here:

In line 15, we create the object `annesAcc` from the `Account` class. The object `annesAcc` is linked to the `Account` class, which provides the three instance methods mentioned above.

In line 16, we add an attribute called balance to the object `annesAcc`, and give it a value of 200. The attribute is only available in this particular object, not to the `Account` class or any other objects which we might create from the `Account` class later on.

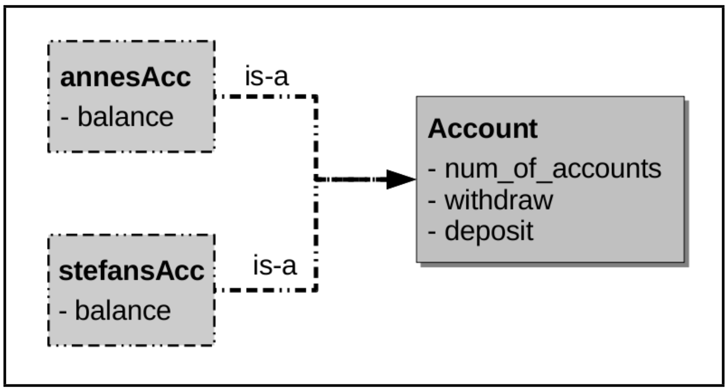
In line 17, we call an instance method of the object `annesAcc`. The object was created from the `Account` class, so we look for the method being called in this class. The method was defined as `deposit(self, amount)`, but we pass just one parameter when calling it (`amount`). Python automatically associates the `annesAcc` object with the parameter `self`, and we only need to provide the parameters from the second one in the parameter list on. Inside the `deposit` method, we have access to the object on which it was called (`annesAcc`) via the `self` variable, and can modify the balance attribute of this object.

Classes can have attributes, too

We already know that classes are some kind of objects, too. Instances link to the class object of the class from which they were created, so for example we can use the `deposit(self, amount)` method of the `Account` class to modify any instance created from the `Account` class. A class object can also have attributes, which we call *class attributes*. In the following example, the `Account` class has a class attribute called `num_of_accounts`, which records how many accounts were created.

So what exactly happens here? In line 4, we create the class attribute by assigning a variable called `num_of_accounts` in the namespace of the `Account` class. Note that the class variable has the same indent as the methods we defined within the class. At the time when the class object is created, the initial value for the class attribute is set to 0.

In the constructor method, we modify the `num_of_accounts` class attribute. Recall that the `__init__` method is called each time when an instance object is created from the class. Each time this happens, we increment the `num_of_accounts` class attribute. We access the class attribute by using the class name together with the dot notation: `Account.num_of_accounts` (as shown in lines 10, 16 and 20).



```python
class Account:
    ''' a class representing an account '''
    # class attributes
    num_of_accounts = 0
    # CONSTRUCTOR
    def __init__(self, num, person):
        self.balance = 0
        self.number = num
        self.holder = person
        Account.num_of_accounts += 1
    # METHODS
    ...

# Main part of the program
if __name__=="__main__":
    print(Account.num_of_accounts, "accounts have been created.")
    annesAcc = Account(1, "Anne")
    annesAcc.deposit(200)
    stefansAcc = Account(2, "Stefan")
    print(Account.num_of_accounts, "accounts have been created."
```

The image on the previous page shows that the `num_of_accounts` attribute resides within the `Account` class object. So far, we have seen the concept: Class objects can have attributes, too, and we assign them in the namespace of the class (either in the class with the correct indent or using `ClassName.attributeName`). However, we can also access the class attributes via instance objects that were created from the class, as the interactive session shows:

```
>>> Account.num_of_accounts
2
>>> annesAcc.num_of_accounts
2
>>> stefansAcc.num_of_accounts
2
```

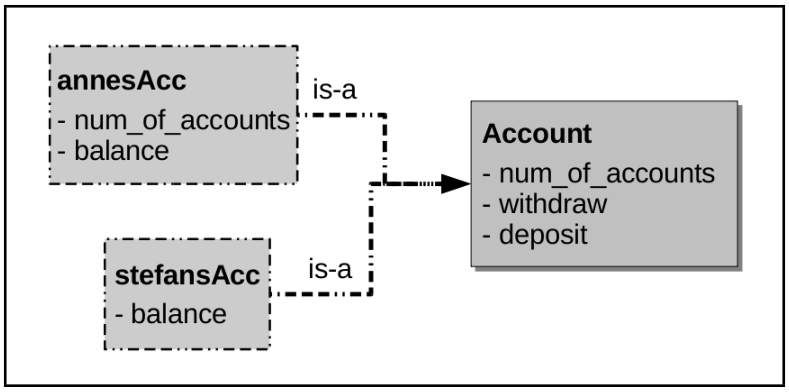
The image on the previous page also explains why that is. As before, Python starts searching for an attribute in the instance object, if it doesn't find it, it moves on the class from which the object was created. Hence, all of the above statements access the same attribute (the class attribute of the `Account` class). If it is changed (e.g. by creating yet another account object), we can see the following output:

```
>>> andreasAcc = Account(3, "Andrea")
>>> Account.num_of_accounts
3
>>> annesAcc.num_of_accounts
3
>>> stefansAcc.num_of_accounts
3
```

So far, so good. In the following case, however, it gets tricky and we have to program carefully in order to avoid bugs.

```
>>> annesAcc.num_of_accounts = 99
>>> Account.num_of_accounts
3
>>> annesAcc.num_of_accounts
99
>>> stefansAcc.num_of_accounts
3
```

In line 1, we assign a *new* instance attribute to the `annesAcc` object. It just happens to have the same name as the class attribute. The following image shows that now, the `annesAcc` object has an attribute called `num_of_accounts` (this attribute happens to have the value 99). However, the class attribute still resides in `Account` class object, and has the value 2. When trying to access the `num_of_accounts` attribute of the instance objects or the class object, we now get different results. While `Account.num_of_accounts` and `stefansAcc.num_of_accounts` still refer to the class attribute, `annesAcc.num_of_accounts` refer to the new instance attribute of this object.



An easy way to avoid this pitfall is to give distinct names to your class attributes and never use the same names for instance attributes.

Static methods belong to a class

So far, we have only seen *instance methods*, which we define inside a class, but which we can only call 'on an object', i.e. when a particular instance is involved. A class can also have methods that do not involve particular instances, but which are called on the class object (e.g. see line 12). Such methods are called *static methods*.

When defining a static method, we need to write `@staticmethod` in the line before the `def` statement for the respective method. `@staticmethod` is a *decorator*. For now, you can imagine that it is a special message you pass to Python, here telling it that the method you are defining belongs to the class (not to particular objects created from the class). The method also does not have the `self` parameter like the instance methods we have seen before.

```
class Account:
    ''' a class representing an account '''
    # class attributes
    num_of_accounts = 0
    ...
    @staticmethod
    def accounts_info():
        print(Account.num_of_accounts, "accounts have been created.")

if __name__=="__main__":
# call a static method
    Account.accounts_info()
```

We can also call class methods on objects of the class, for instance `annesAcc.accounts_info()`. In this case, Python calls the `accounts_info()` method of the class from which `annesAcc` was created. It does not pass `annesAcc` as an implicit argument as it is done in instance methods using the `self` parameter. As you can see, static methods don't care about particular objects, and the coding style is better if you call them on the class as in the listing above.

Python also provides a more sophisticated way to define methods that belong to classes, so-called *class methods*. These are beyond the scope of this tutorial.

Using classes in other files

Usually, you will write your classes into various files (modules), and you want to use them in another file which contains your main application. An easy way to be able to use your class is to write the class definition into a file (module) which we call *accounts.py*, and then have the main part of your application in another file, here called *bank.py*. The *bank.py* file must then import your classes. Note that the syntax for importing your class is `from modulename import classname`, where `modulename` has to match the file name in which your class is defined (minus the `.py`) and `classname` is the name of your class.

```
from accounts import Account

if __name__ == "__main__":
    annesAcc = Account()
    annesAcc.balance = 200
    annesAcc.deposit(500)
    annesAcc.withdraw(20)
    annesAcc.print_info()
```

Composition/Aggregation

Recall that each value in Python has a type, e.g. `1.5` has the type *float* or `'python'` has the type *str* (=string). If we create an object, it also has a type, which is the class from which it was created. We can check the type of an object using the `type` function:

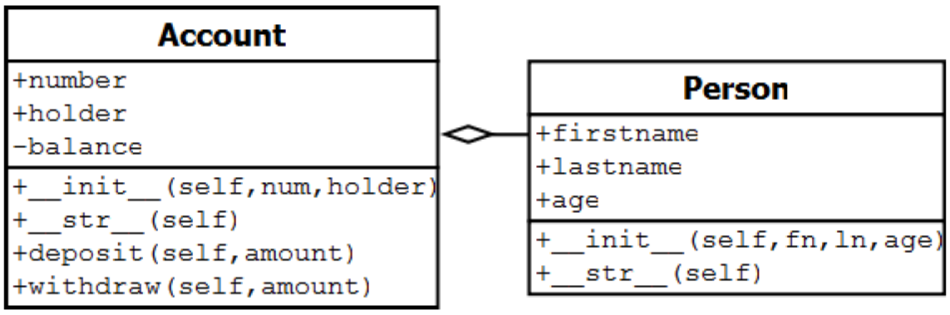
```
>>> stefansAcc = Account(2, "Stefan")
>>> type(stefansAcc)
<class '__main__.Account'>
```

The type of the `stefansAcc` object is the class from which it was created.

The attributes of an object can be of any type. This means that the attributes of an object can be objects themselves. In fact, although we haven't told you so far, anything is an object in Python, also strings, numbers, lists or dictionaries. The terms *composition* and *aggregation* refer to the fact that we can compose more complex objects out of multiple objects. Composition implies that the owned objects only exist 'within' the more complex object. For instance, if a university is closed, so are all its departments. Aggregation implies that an object is part of a more complex object, but if the larger object is destroyed, the smaller one may still exist. For instance, we can define a `Person` class that deals with information about persons, and we save the information about an account holder in a `Person` object as shown in the following example. If the account is closed, the person still exists (and so may its object - a bank might save information about a former customer even after the customer closed his or her account).

```python
class Person:
    def __init__(self, f, l, a):
        self.firstname = f
        self.lastname = l
        self.age = a
    def __str__(self):
        return "[Person: " + self.firstname + " " + \
               self.lastname + " (" + str(self.age) + ")]"

class Account:
    def __init__(self, person, num):
        self.holder = person
        self.num = num
        self.balance = 0
    def deposit(self, amount):
        self.balance += amount

anne = Person("Anne", "Friedrich", 85)
annesAcc = Account(anne, 1)
```

In UML class diagrams, we can show that one class has an attribute whose type is another class using a line that connects the two classes. The diamond symbol shows that the `Account` class *uses* the `Person` class. In the case of composition, the diamond symbol is filled (black).



Please note that UML class diagrams show the *class design*, i.e. how objects created from this class will look like. They show the attributes and methods that will be available for objects created from this class. UML diagrams do not show what is going on in Python internally (for this we used the diagrams with the grey boxes). For example, instance attributes will not be available in the class object, but we list them in the class diagram. Class attributes and static methods are underlined in UML class diagrams.

Inheritance makes a programmer's life easier

Inheritance is a central principle in OOP which leverages commonalities and differences between objects which have to be dealt with in our code. The big advantage, as we will see, is that inheritance minimizes *redundancy* (we don't have to write the same code over and over again) and thus also facilitates maintenance (if we need to change something in the code, we need to do it only once, if the functionality is shared by different classes). This may sound very abstract, so let's have a look at a concrete example.

Assume our bank offers two different kind of accounts:

- **Savings Account**: We record the account number, holder and balance with each account. The balance has to be ≥ 0. We can apply an interest rate which is defined once for all savings accounts. Money can be deposited into the account. The account statement that can be printed includes the account number, holder and balance.
- **Checking Account**: We record the account number, holder and balance with each account. The balance has to be greater than or equal to a credit range which is determined on a per-customer basis. For instance, if the credit range determined for Anne is $500, her balance may not be less than - $500. Money can be deposited into the account. The account statement that can be printed includes the account number, holder and balance.

Usually, the development of an application starts with a description of some real-life objects (such as the accounts) like above. Now, we have to map these descriptions into an object-oriented design. We can make the following statements about our application setting:

1. Both the savings account and the checking account are some type of account.
2. We can deposit money into either account type.
3. The account statements to be printed out are the same for the two account types.
4. The savings account and the checking account both have the following attributes:
    - account number
    - account holder
    - balance
5. The savings accounts have an additional parameter, the interest rate, but which is the same for all savings accounts.
6. Savings accounts have a behavior that checking accounts don't have: We can apply the interest rate on their balances.
7. Each checking account has its own particular credit range.
8. Cash withdrawal works differently for the two account types. In the savings account, the balance may not be less than 0, while for checking accounts, the balance may not be less than the credit range defined for the particular client.

Base classes provide general functionality

All of the above considerations will be reflected in our class design. First of all, we will code a *base class* which implements all the things that are shared by the two types of accounts: the attributes holder, number and balance, methods for deposit and withdrawal as well as for printing out the state of the account. The following listing shows our base class, `Account`, which doesn't look much different from the previous class. In fact, we could instantiate objects directly from this class and use them as accounts.

```python
class Account:
    ''' a class representing an account '''
    # CONSTRUCTOR
    def __init__(self, num, person):
        self.balance = 0
        self.number = num
        self.holder = person
    # METHODS
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        if amount > self.balance:
            amount = self.balance
        self.balance -= amount
        return amount
    def __str__(self):
        res = "*** Account Info ***\n"
        res += "Account ID:" + str(self.number) + "\n"
        res += "Holder:" + self.holder + "\n"
        res += "Balance: " + str(self.balance) + "\n"
        return res
```

Derived classes provide for special needs

Let's turn to the savings accounts. In fact, they are a specialized case of the `Account` class we have just written. Next, we are going to write a class called `SavingsAccount` which *extends* the `Account` class, or is *derived* from it. This basically means that all functionality that is available in the `Account` class is also available in the `SavingsAccount` class. We tell Python that the `SavingsAccount` class is based on the `Account` class by starting the class definition with `class SavingsAccount(Account).` We say that a *derived class/subclass* is *based* on a *superclass/base class*. This means that anything that is available in the superclass is also available in the subclass.

```python
class SavingsAccount(Account):
    ''' class for objects representing savings accounts.
        shows how a class can be extended. '''
    interest_rate = 0.035
    # METHODS
    def apply_interest(self):
        self.balance *= (1+SavingsAccount.interest_rate)
```

In our main application, we instantiate two savings accounts.

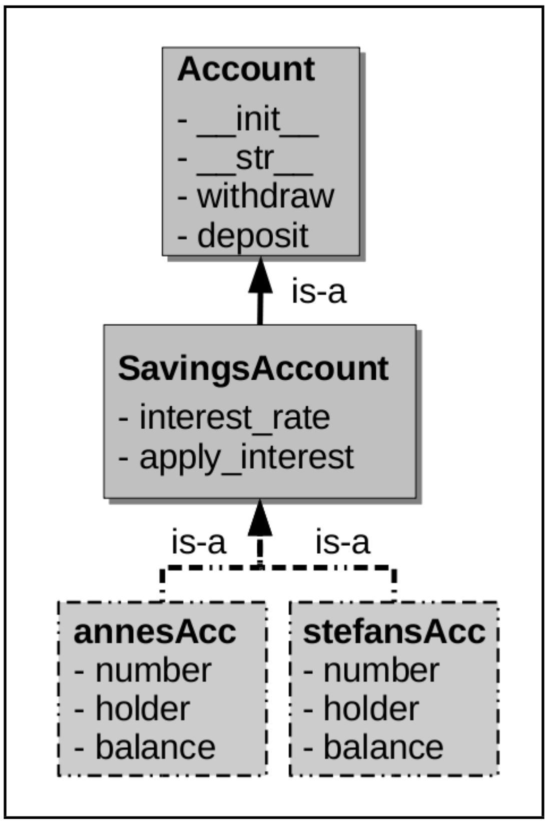
```python
if __name__=="__main__":
    annesAcc = SavingsAccount(1, "Anne")
    annesAcc.deposit(200)
    annesAcc.apply_interest()
    print(annesAcc)

    stefansAcc = SavingsAccount(2, "Stefan")
    stefansAcc.deposit(1000)
    stefansAcc.apply_interest()
    print(stefansAcc)
```

The image on the next page shows the *class hierarchy* and the objects we instantiated from it.

Note that although the instance attributes balance, number and holder were set in the constructor of the `Account` class, they are attributes of the instance objects. We set them as `self.balance`, not as `Account.balance` in the constructor method.

When we apply any instance method on an object, the class hierarchy is searched from bottom to the top. For instance, we did not define a constructor method in the `SavingsAccount` class. Because the `SavingsAccount` class is based on the `Account` class, Python continues looking for an `__init__` method at the superclass (`Account`), finds it there and executes this method when creating a new object by calling the `SavingsAccount` class.

Even simpler, when calling `annesAcc.deposit(200)`, Python starts looking for a `deposit` method at the `annesAcc` object, but doesn't find one there. It then looks at the class from which the object was created, which happens to be `SavingsAccount`. It doesn't find the method there either, so it continues looking for the class in the superclass of `SavingsAccount`, `Account`, where it finally finds the method. The `apply_interest(self)` method is found at the `SavingsAccount` class, which means that the search stops here and executes this method.

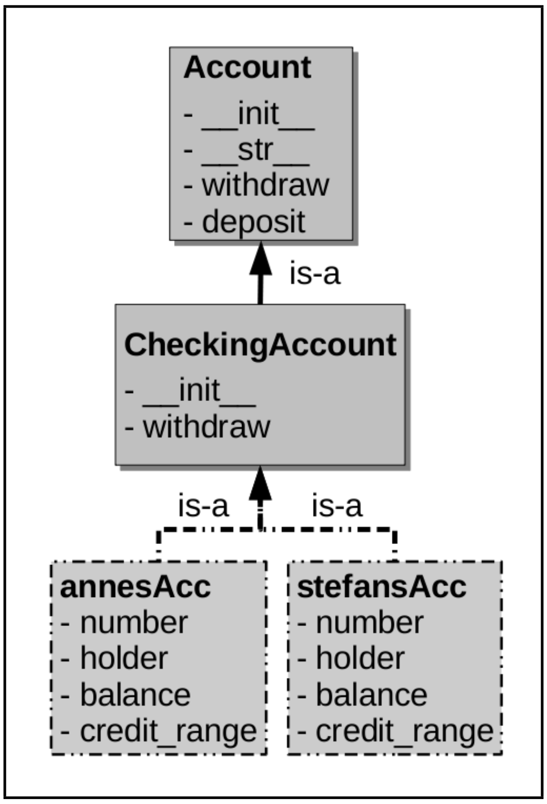


Subclasses can override the behavior of their superclasses

Now, we define a class called `CheckingAccount` which is also based on the `Account` class. However, the general `withdraw(self, amount)` method, which we defined in the `Account` class and which does not allow a withdrawal to result in a negative balance, does not apply to checking accounts. Recall that a checking account may have a negative balance, but only up to a credit range which is defined on a per-customer basis. The following listing shows how we can redefine/replace, or (as in OOP-speak) *override* methods of a superclass in a subclass.

```python
class CheckingAccount(Account):
    ''' class for objects representing checking accounts.
        shows how methods can be overridden '''
    # CONSTRUCTOR
    def __init__(self, num, person, credit_range):
        print("Creating a checkings account")
        self.number = num
        self.holder = person
        self.balance = 0
        self.credit_range = credit_range
    # METHODS
    def withdraw(self, amount):
        amount = min(amount, abs(self.balance + self.credit_range))
        self.balance -= amount
        return amount
```

```python
stefansAcc = CheckingAccount(2, "Stefan", 2000)
stefansAcc.deposit(1000)

annesAcc = CheckingAccount(1, "Anne", 500)
annesAcc.deposit(200)
annesAcc.withdraw(350)
print(annesAcc)
print("trying to withdraw 400")
cash = annesAcc.withdraw(400)
print("Got only: ", cash)
print(annesAcc)
```



As we can see, the `CheckingAccount` class provides a constructor method (__init__). When creating an object by calling the `CheckingAccount` class, Python starts looking for a constructor at this class and immediately finds one. It executes the `__init__` method of the `CheckingAccount` class and sets the four attributes of the object being created. The constructor method of the superclass `Account` is not executed in this case.
The `CheckingAccount` class also overrides the `withdraw(self, amount)` method of its superclass. Overriding a method simply works by giving the exact same name to a method. When we call this method on the `annesAcc` object, Python again starts looking for the method in the object and then at the classes it is linked to from bottom to top. It finds the method at the `CheckingAccount` class and executes this method.

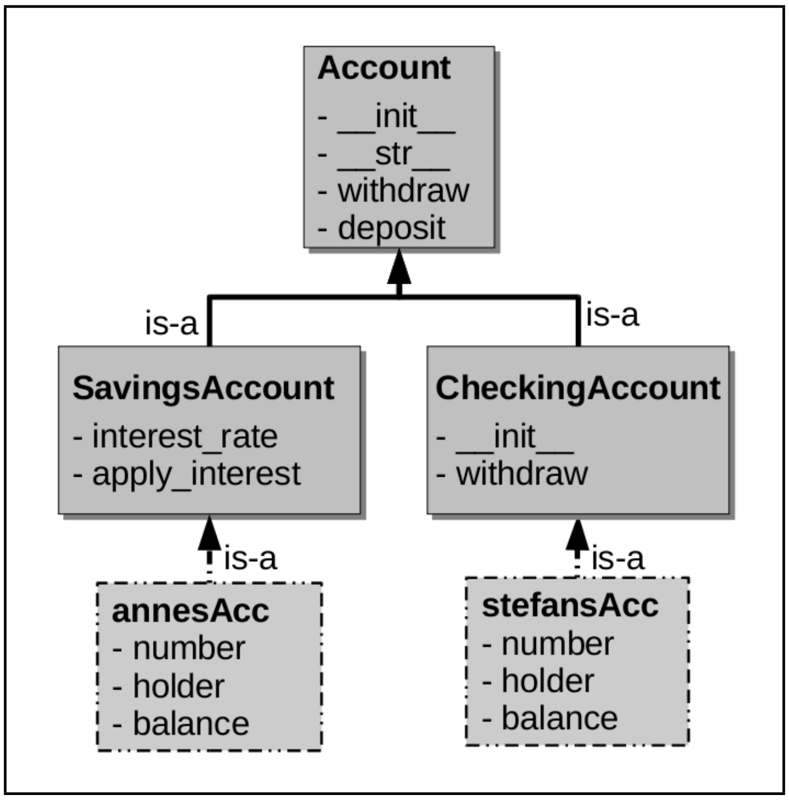
Methods with the same name can do different things

We have defined a *class hierarchy* by now, `SavingsAccount` and `CheckingAccount` are both subclasses of `Account`. Let us create two objects of these classes as follows:

```python
annesAcc = SavingsAccount(1, "Anne")
annesAcc.deposit(200)
annesAcc.withdraw(350)
stefansAcc = CheckingAccount(2, "Stefan", 500)
stefansAcc.deposit(1000)
stefansAcc.withdraw(300)
```

The image below shows the complete class hierarchy and our two account objects. When creating `annesAcc`, Python looks for the constructor method in the `SavingsAccount` class, doesn't find on, and moves on to the `Account` class. It then modifies the newly created object as coded in the `__init__` method of the `Account` class. When creating the `stefansAcc` object, Python finds a constructor method in the `CheckingAccount` class and uses this constructor directly to modify the newly created object.

When we apply the `deposit` method on either the `annesAcc` or the `stefansAcc` object, Python will execute the `deposit` method of the `Account` class, because it doesn't find a `deposit` method anywhere lower in the tree.



When we apply the `withdraw` method on the `stefansAcc` object, again, Python finds the method definition in the `CheckingAccount` class and executes this method. When we apply `withdraw` on `annesAcc`, Python executes the `deposit` method defined in the `Account` class.

We just saw an example of *polymorphism*. The word *polymorphism* is derived from the Greek word for 'having multiple forms'. In this case, it means that we can call a method that has the same name (for instance `withdraw`) on several objects (here `annesAcc` and `stefansAcc`), and what happens depends on the inheritance hierarchy of the classes from which the objects were created. The nice thing here is, at the point of time when we do the withdrawal, we simply say `accountObject.withdraw(amount)` and we don't need to care about whether the `accountObject` is a savings or a checking account. Python knows which inheritance lines to follow and the desired behavior is produced in either case.

Of course, we could simply have defined the `withdraw` method twice, once in the `SavingsAccount` class and once in the `CheckingAccount` class. We defined the method in the superclass here in order to give an example for overriding methods.

This might even make sense if our bank provides many more types of accounts and in most of these accounts, the withdrawal works as defined in the `Account` class. Then, most of the classes derived from `Account` simply default to this behavior, while we can provide special behavior for checking accounts.

Subclasses can extend functionality of superclasses

We said that OOP is great because it helps us to minimize redundancy and thus makes code maintance easier. Minimizing redundancy means to never write the same piece of code twice. Compare the constructor methods of our `Account` class and the `CheckingAccount` class, which is derived from it. Here, we just copied and pasted parts of what is going on inside the constructor method of `Account` into the constructor method of `CheckingAccount`.

```
class Account:
    ''' a class representing an account '''
    # CONSTRUCTOR
    def __init__(self, num, person):
        self.balance = 0
        self.number = num
        self.holder = person
    # METHODS
   ...

class CheckingAccount(Account):
    ''' class for objects representing checking accounts.
        shows how methods can be overridden '''
    # CONSTRUCTOR
    def __init__(self, num, person, credit_range):
        self.number = num
        self.holder = person
        self.balance = 0
        self.credit_range = credit_range
    # METHODS
    ...
```

As OOP is great, there is of course a better way to do it. We can call the constructor of a superclass in the constructor of a subclass. In this particular case, it looks like this:

```
class CheckingAccount(Account):
    ''' class for objects representing checking accounts.
        shows how methods can be overridden '''
    # CONSTRUCTOR
    def __init__(self, num, person, credit_range):
        Account.__init__(self, num, person)
        self.credit_range = credit_range
    # METHODS
   ...
```

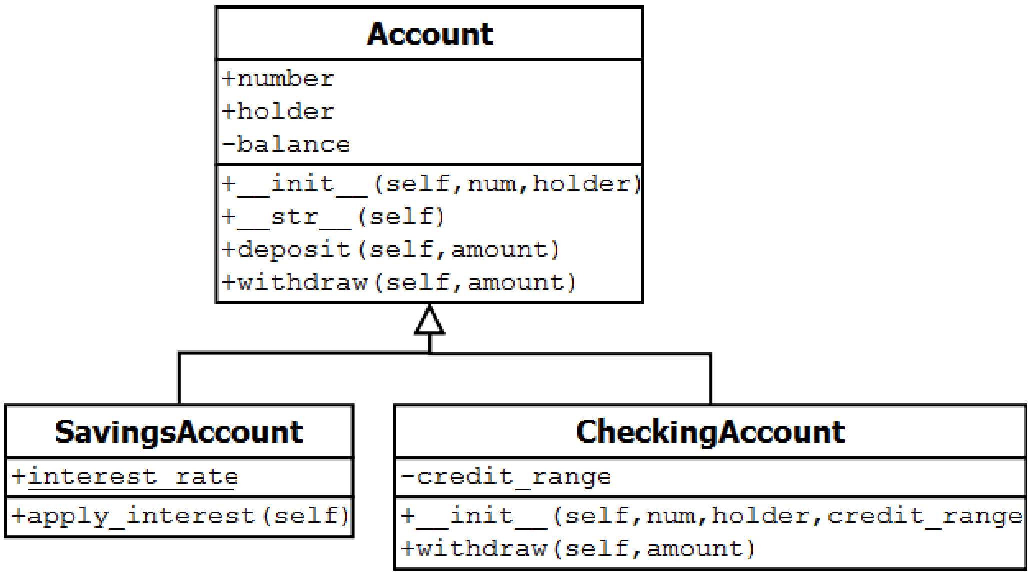
In line 6, we call the constructor method of the `Account` class. Here, we need to explicitly pass on the `self` parameter, because we are not calling the method 'on an object', but 'on the class'. By passing on the `self` parameter, Python will know which object to operate on when it executes the constructor method of the `Account` class.

In general, we can call any method of a superclass in this way, not only the constructor method. For instance, we could implement the `withdraw` method in a very general way in the base class, and do all the consistency checking in the subclasses, as shown in the following example. (In this case, we're not really saving lines of code, but in a more complex case, the method in the superclass might be more complex, and then it really makes sense to structure your code like this.)

```python
class Account:
    def withdraw(self, amount):
        self.balance -= amount

class SavingsAccount(Account):
    # METHODS
    def withdraw(self, amount):
        ''' balance must be > 0 '''
        if amount > self.balance:
            amount = self.balance
        cash = Account.withdraw(self, amount)
        return cash

class CheckingAccount(Account):
    ''' class for objects representing checking accounts.
        shows how methods can be overridden '''
    # CONSTRUCTOR
    def __init__(self, num, person, credit_range):
        Account.__init__(self, num, person)
        self.credit_range = credit_range
    # METHODS
    def withdraw(self, amount):
        ''' balance must be > credit range '''
        amount = min(amount,
             abs(self.balance + self.credit_range))
        cash = Account.withdraw(self, amount)
        return cash
```

UML Class Diagrams: Inheritance

In UML class diagrams, inheritance is shown by connecting the classes with a line. The triangle points to the superclass. It is convention to put the superclass on top of its subclasses (if space allows). Recall that class attributes and static methods are underlined in UML class diagrams (such as the class attribute `interest_rate` in the example below.



Classes can have multiple base classes

Unlike other object-oriented programming languages such as Java, Python allows *multiple inheritance*. It means that a base class can be derived from more than one class. When using multiple inheritance, you must be aware of some special mechanisms (which we are not going to explain here) that define which class's method is called. Unless you are an experienced programmer, we recommend that you try not to use this feature and always base your class on a single class instead.

A few remarks on terminology

For beginners, OOP may seem like extra coding efforts that are not necessary in some cases. So why is it essential to understand OOP if you want to be a good programmer these days? The main reason is that you will rarely program *from scratch*, but by using, extending and customizing other people's code. Particularly, you will use *frameworks* which are basically collections of superclasses that implement common programming tasks. Often, you will just have to write a subclass for one of the framework's classes that specializes the behavior for you specific application. In fact, extensions in this way are so common that *design patterns* have been developed that are recipes for good ways to extend or leverage classes.

Another issue of confusion is the term *data encapsulation*, because it is used for two aspects of OOP. The first, which we introduced in this tutorial, is that data should be hidden, i.e. only accessed via instance methods, in order to make sure that our object is in a valid state. The other aspect of OOP that is often described using the term *encapsulation* is that OOP helps to wrap up program logic behind interfaces ($\approx$ class names and the methods of the functions) such that each functionality is only defined once in a program. When you work with the functionality, you just call the respective class's method and you don't care what happens 'under the hood'. In fact, you can even change the implementations that happen 'under the hood' without changing the code that uses the interface. This can be useful for instance if you suddenly think of a more efficient implementation of a particular function. All you need to change is exactly this function.

References

[1] Mark Lutz: *Learning Python*, Part VI, 4th edition, O'Reilly, 2009.

[2] Michael Dawson: *Python Programming for the Absolute Beginner*, Chapters 8 & 9, 3rd edition, Course Technology PTR, 2010.