Python advanced class

Module 5, Classes and objects

- Object oriented design
- Defining a class
- Instantiation
- Dataclasses
- Operator overloading
- Polymorphism
- Inheritance
- Property

Object oriented design

The model of a phenomena in the real World

- Most computer programs model some part of things in the real World:
    - Bank account
    - Bet on a race horse
    - A seat in a theater on a particular play night
- These phenomenas has a set of properties and some operations on them
- The idea behind Object Oriented Programming is to model things we need to represent

Attributes and Methods

- If we want to represent a bank account it will have some attributes and methods to interact with it
- Attributes could be the balance, the contact info of the owner, and the maximum withdrawal at a time
- The methods could be to deposit more money, withdraw money, request a balance statement, or close it
- Some objects has several other objects embedded like a bank has several accounts, a theater has several seats
- Some attributes are related to a single object like the saldo of an acount, while other relate to all accounts like currency

Class vs. Instance/Object

- A class is a representation of a type or a concept like bank accounts
- When a model changes from a type to a particular one we say that we go from class to object
- That process is called instantiation and is forming an object from a set of properties into attributes of the object
- Everything in Python is an object, and every object has a type or class, even classes are objects

Defining a class

The class

- A class is defined as a name and optionally another class it inherits from (the class `object` if not specified)
- After the name comes a set of class attributes and methods related to instances of the class

In [1]:
```python
class BankAccount:
    currency: str = "EUR"

    def deposit(self, amount):
        pass
```

The `__init__` method

- When an object has been created it should most likely need to receive some data, attributes
- The safest and easiest way to ensure this is to define an initialization method, the `__init__` method
- This sets the attributes the moment the object is created so it never will exist without its data

In [2]:
```python
class BankAccount:

    def __init__(self, initial_balance, owner):
        self.balance = initial_balance
        self.owner = owner
```

The `__str__` method

- It is common to want to display an object with key attributes
- If the class defines a `__str__` method, it can just be printed

In [3]:

```python
class BankAccount:

    def __init__(self, initial_balance, owner):
        self.balance = initial_balance
        self.owner = owner

    def __str__(self):
        return f"BankAccount(balance={self.balance}, owner={self.owner!
```

Instantiation

From class to object

- If we want to have an object of the type BankAccount we must instantiate the class
- This is like calling a function that returns an object
- The intantiation must provide the arguments to the initialization method

In [4]:

```python
class BankAccount:

    currency: str = "EUR"

    def __init__(self, initial_balance, owner):
        self.balance = initial_balance
        self.owner = owner

    def __str__(self):
        return f"BankAccount(balance={self.balance}, owner={self.owner!

    def deposit(self, amount):
        self.balance += amount

savings = BankAccount(10000, "Ben Birch")
print(savings)
```

```
BankAccount(balance=10000, owner='Ben Birch')
```

Accessing attributes and methods

- The attributes and methods are accessed from the instance object by at dot, `.`
- The `self` variable seen earlier is the actual instance

In [5]:

```python
savings = BankAccount(10000, "Ben Birch")
print(savings)
savings.deposit(500)
print(savings.balance)
```

```
BankAccount(balance=10000, owner='Ben Birch')
10500
```

isinstance function

- The proper way to determine if an object is and instance of a particular class is the `isinstance` function
- But more often we don't need to know the class but rather if the object conforms to operate like objects of the class

In [6]:

```
savings = BankAccount(10000, "Ben Birch")
if isinstance(savings, BankAccount):
    print("It's a bank account")
```

```
It's a bank account
```

Dataclasses

Dataclasses makes life easier

- Many operations related to checking datatypes making string function can be simplified with dataclasses
- Most classes can easier be defined by using dataclasses

In [7]:
```python
from dataclasses import dataclass, field

@dataclass
class BankAccount:
    balance: int
    owner: str

    def deposit(self, amount):
        self.balance += amount

savings = BankAccount(10000, "Ben Birch")
print(savings)
```

```
BankAccount(balance=10000, owner='Ben Birch')
```

Declaring attributes

- Attributes can just be declared immediately after the class declaration
- If an attribute has a reasonable default value it can be set
- If an attribute of a mutable type must be initiated the `field` function must be used

In [8]:
```python
@dataclass
class BankAccount:
    balance: int = 0
    owner: str = None
    transactions: list = field(default_factory=list)

savings = BankAccount(10000, "Ben Birch")
print(savings.transactions)
```

```
[]
```

The `__post_init__` method

- If additional operations are necessary after the initialization a `__post_init__` method can be used

In [9]:
```
@dataclass
class BankAccount:
    balance: int = 0
    owner: str = None

    def __post_init__(self):
        if self.balance < 0:
            raise ValueError("Can't create an account with negative bala

try:
    savings = BankAccount(-10000, "Ben Birch")
except ValueError as e:
    print("The account could not be made:", e)
else:
    print(savings.transactions)
```

```
The account could not be made: Can't create an account with nega
tive balance
```

Operator overloading

Arithmetic operators

- Many types in Python allows adding them with the `+` operator: int, float, str, list, ...
- Two instances of a class can be added, if the class implements the `__add__` method
- Additionally there is `__sub__`, `__mul__`, `__div__`

In [10]:
```python
@dataclass
class BankAccount:
    balance: int = 0
    owner: str = None

    def __add__(self, other):
        if self.owner != other.owner:
            raise ValueError("Can't add accounts owned by different peop
        return BankAccount(balance = self.balance + other.balance, owne

savings = BankAccount(10000, "Ben Birch")
checking = BankAccount(1500, "Ben Birch")
one_account = savings + checking
print(one_account)
```

```
BankAccount(balance=11500, owner='Ben Birch')
```

Comparison operators

- If two objects should be comparable for one larger than the other, the `__gt__` must be implemented
- There is likewise `__lt__`, `__ge__`, `__le__`, `__eq__`, `__ne__`, `__contains__`

In [11]:
```python
@dataclass
class BankAccount:
    balance: int = 0
    owner: str = None

    def __gt__(self, other):
        return self.balance > other.balance

savings = BankAccount(10000, "Ben Birch")
checking = BankAccount(1500, "Ben Birch")
print(savings > checking)
```

```
True
```

Polymorphism

Duck typing

- In other languages than Python it is not possible to create lists of unrelated objects
- In Python the strict type checking is done for that, as long as the classes implements methods and attributes need
- This is called duck-typing

In [12]:
```
class Duck:
    def walk(self):
        print("Walks like a duck")

class ScubaDiver:
    def walk(self):
        print("Walks almost like a duck")

L1 = [Duck(), Duck(), ScubaDiver(), Duck()]
for thing in L1:
    thing.walk()
```

```
Walks like a duck
Walks like a duck
Walks almost like a duck
Walks like a duck
```

A list of shapes

- Imagine a drawing program with different shapes
- When all shapes in the canvas needs re-drawing, each shape can draw itself

In [13]:
```python
class Triangle:
    pass
class Square:
    pass
class Circle:
    pass

shapes = [Triangle(), Triangle(), Circle(), Square(), Circle()]
for s in shapes:
    pass # Do stuff with each shape
```

Inheritance

A derived class

- A derived class is a specialized class based on a base class
- This could be a bank account for a loan with fixed payments

In [14]:
```
class BankAccount:
    pass

class LoanAccount(BankAccount):
    pass

loan = LoanAccount()
print(isinstance(loan, BankAccount))
```

```
True
```

The super function

- A derived class will have most of the same functionality of the base class
- It is often practical to reuse most of the base class methods
- A `super` function will call the base class method of the same name

In [15]:
```
class BankAccount:
    def __init__(self, balance, owner):
        self.balance = balance
        self.owner = owner

class LoanAccount(BankAccount):
    def __init__(self, balance, owner, payment):
        super().__init__(balance, owner)
        self.payment = payment

loan = LoanAccount(-10000, "Dennis Dawson", 1000)
print(loan.owner, loan.payment)
```

```
Dennis Dawson 1000
```

Property

Hiding logic behind a property

- A property is a virtual attribute provided by a set of functions

In [16]:
```python
class Person:
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

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

    def set_fullname(self, name: str):
        parts = name.split(" ")
        self.firstname = parts[0]
        self.lastname = parts[1]

    fullname = property(get_fullname, set_fullname)

p = Person("Dennis", "Dawson")
print(p.fullname)
p.fullname = "Denis Dawsson"
print(p.firstname)
```

```
Dennis Dawson
Denis
```