Max Planck Institute for Mathematics in the Sciences

ScaDS.AI DRESDEN LEIPZIG

Intro. Comp. for Data Science (FMI08)

Dr. Nono Saha

July 5, 2023

Max Planck Institute for Mathematics in the Sciences
University of Leipzig/ScaDS.AI

Spring 2023

Course plan

1. Class, interface, abstract class and object
2. Iterable objects
3. Generators
4. Relationship between classes
5. Polymorphism

Why OOP in Python?

Classes are the building blocks of object-oriented programming (OOP) in Python. They allow you to leverage the power of Python while writing and organizing your code.

- Model and solve complex real-world problems
- Reuse code and avoid repetition
- Encapsulate related data and behaviours in a single entity
- Abstract away the implementation details of concepts and objects
- Unlock polymorphism with common interfaces

Class, interface, abstract class and object

Basic syntax

These are the basic component of Python's object-oriented system - we've been using them regularly all over the place and will now look at how they are defined and used.

```
class Rectangle:
  """An abstract representation of a rectangle"""
  # Attributes
  p1 = (0,0)
  p2 = (1,2)

# Methods
def area(self):
  return abs(self.p1[0] - self.p2[0]) *abs(self.p1[1] - self.p2
    [1])

# Setters
def setP1(self, p1):
  self.p1 = p1
def setP2(self, p2):
  self.p2 = p2
```

Interfaces and abstract classes

Two important points

- Interfaces are classes that contain methods without implementations
- Abstract classes are classes with at least one method without implementation

Example

```
1  class AbstractRectangle(abc.ABC) :
2
3      def __init__(self, p1=(0,0), p2=(1,2)) :
4        self.p1 = p1
5        self.p2 = p2
6
7      @abc.abstractmethod
8      def area(self):
9        pass
10
```

Objects

What is an object?

- Objects are instances of a class
- Objects can also represent different states of a class
- Objects are coherent entities that store data and the code (or instructions) working on that data.

Example

```
x = Rectangle()
x.area()

```

```
y = Rectangle(p2= (5,4))
y.area()
```

Class attributes

We can examine all of a classes' methods and attributes using `dir()`,

```
dir(Rectangle)
## ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__
    ', '__eq__', '__format__', '__ge__', '__getattribute__', '
    __gt__', '__hash__', '__init__', '__init_subclass__', '
    __iter__', '__le__', '__lt__', '__module__', '__ne__', '
    __new__', '__reduce__', '__reduce_ex__', '__repr__', '
    __setattr__', '__sizeof__', '__str__', '__subclasshook__',
    '__weakref__', 'area']
```

Where did p1 and p2 go?

```
dir(Rectangle())
## ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__
    ', '__eq__', '__format__', '__ge__', '__getattribute__', '
    __gt__', '__hash__', '__init__', '__init_subclass__', '
    __iter__', '__le__', '__lt__', '__module__', '__ne__', '
    __new__', '__reduce__', '__reduce_ex__', '__repr__', '
    __setattr__', '__sizeof__', '__str__', '__subclasshook__',
    '__weakref__', 'area', 'p1', 'p2']
```

Instantiation (constructors)

When instantiating a class (e.g. Rectangle()) we invoke the `__init__()` method if it is present in the classes' definition.

```
class Rectangle:
"""An abstract representation of a rectangle"""
# Constructor
def __init__(self, p1 = (0,0), p2 = (1,1)):
  self.p1 = p1
  self.p2 = p2

# Methods
def area(self):
  return ((self.p1[0] - self.p2[0])*(self.p1[1] - self.p2[1]))
def setP1(self, p1):
  self.p1 = p1
def setP2(self, p2):
  self.p2 = p2
```

Method chaining

We've seen a number of objects (i.e. Pandas `DataFrames`) that allow for method chaining to construct a pipeline of operations. We can achieve the same by having our class methods return self.

```
class Rectangle:
"""An abstract representation of a rectangle"""
# Constructor
def __init__(self, p1 = (0,0), p2 = (1,1)):
  self.p1 = p1
  self.p2 = p2

# Methods
def area(self):
  return ((self.p1[0] - self.p2[0]) *(self.p1[1] - self.p2[1]))
def setP1(self, p1):
  self.p1 = p1
  return self
def setP2(self, p2):
  self.p2 = p2
  return self
```

Object string formating

All class objects have a default print method / string conversion method, but the default behavior is not very useful,

```
print(Rectangle())
## <__main__.rect object at 0x290aa1a60>

str(Rectangle())
## '<__main__.rect object at 0x290aa1ca0>'
```

Both of the above are handled by the `__str__()` method which is implicitly created for our class - we can override this,

```
def rect_str(self):
  return f"Rectangle[{self.p1}, {self.p2}] => area={self.area()}
    "


Rectangle.__str__ = rect_str
```

Class representation

There is another special method which is responsible for the printing of the object (see Rectangle() above) called `__repr__()` which is responsible for printing the classes representation. If possible this is meant to be a valid Python expression capable of recreating the object.

```
def rect_repr(self):
  return f"Rectangle({self.p1}, {self.p2})"

rect.__repr__ = rect_repr
```

```
Rectangle()
## Rectangle((0, 0), (1, 1))
```

```
repr(Rectangle())
## Rectangle((0, 0), (1, 1))
```

OOP: object relationship

Inheritance

Part of the object-oriented system is that classes can inherit from other classes, meaning they gain access to all of their parent's attributes and methods. It models a **Is a** relationship.

In an inheritance relationship:

- Classes inherited from another are derived classes, subclasses, or subtypes.
- Classes from which other classes are derived are called base classes or superclasses.
- A derived class is said to derive, inherit, or extend a base class.

```
class Square(Rectangle):
  pass
```

```
  Square()
  ## Rectangle((0, 0), (1, 1))
```

Multiple inheritance

A class can be derived from more than one superclass in Python. This is called multiple inheritance.

Example

```
class Worm:
  def __init__ (self, name) :
    self.name = name
  def eat(self):
    print(self.name +" swallows")

class Fly:
  def __init__ (self, name) :
    self.name = name
  def eat(self):
    print(self.name +" is nibbling..")

class ButterFly(Worm, Fly):
  pass
```

Inheritage: overriding methods

```
class Square(Rectangle):
  def __init__(self, p1=(0,0), l=1):
    assert isinstance(l, (float, int)), "l must be a numnber"
    p2 = (p1[0]+l, p1[1]+l)
    self.l  = l
    super().__init__(p1, p2)

  def setP1(self, p1):
    self.p1 = p1
    self.p2 = (self.p1[0]+self.l, self.p1[1]+self.l)
    return self
  def setP2(self, p2):
    raise RuntimeError("Squares take l not p2")
  def setL(self, l):
    assert isinstance(l, (float, int)), "l must be a numnber"
    self.l  = l
    self.p2 = (self.p1[0]+l, self.p1[1]+l)
    return self
  def __repr__(self):
    return f"square({self.p1}, {self.l})"
```

Making an object iterable

When using an object with a for loop, python looks for the `__iter__()` method which is expected to return an iterator object (e.g. iter() of a list, tuple, etc...).

```
class Rectangle:
  """An object representation of a rectangle"""
  # Constructor
  def __init__(self, p1 = (0,0), p2 = (1,1)):
    self.p1 = p1
    self.p2 = p2

  # Methods
  def area(self):
    return ((self.p1[0] - self.p2[0]) *(self.p1[1] - self.p2[1])
    )

  def __iter__(self):
    return iter( [ self.p1, (self.p1[0], self.p2[1]),
    self.p2, (self.p2[0], self.p1[1]) ] )
```

Generators

We can improve the implementation above by simplify a generator function with `__iter__()`. A generator is a function which using `yield` instead of `return` which allows the function to preserve state between `next()` calls.

```
class rect:
  """An object representation of a rectangle"""
  # Constructor
  def __init__(self, p1 = (0,0), p2 = (1,1)):
    self.p1 = p1
    self.p2 = p2
  # Methods
  def area(self):
    return ((self.p1[0] - self.p2[0]) *(self.p1[1] - self.p2[1])
    )
  def __iter__(self):
    vertices = [ self.p1, (self.p1[0], self.p2[1]), self.p2, (
    self.p2[0], self.p1[1]) ]
    for v in vertices:
      yield v
```

Composition in Python

Composition is an OOP concept that models a has a relationship. In composition, a class known as composite contains an object of another class known as a component. In other words, a composite class has a component of another class.

Example: we already used it in the previous example, but how? and where?

Remarks

- Composition is more flexible than inheritance because it models a loosely coupled relationship
- Changes to a component class have minimal or no effects on the composite class
- Designs based on composition are more suitable to change

Example of composition in Python

```
class Salary:
  def __init__(self, pay, bonus):
    self.pay = pay
    self.bonus = bonus

  def annual_salary(self):
    return (self.pay*12)+self.bonus

class EmployeeOne:
  def __init__(self, name, age, pay, bonus):
    self.name = name
    self.age = age

  self.obj_salary = Salary(pay, bonus)  # composition

  def total_sal(self):
    return self.obj_salary.annual_salary()

emp = EmployeeOne('Geek', 25, 10000, 1500)
print(emp.total_sal())
```

Aggregation in Python

Aggregation is a concept in which an object of one class can own or access another independent object of another class.

- It represents **Has-A**'s relationship.
- It is a unidirectional association, i.e. a one-way relationship. For example, a department can have students, but vice versa is not possible and thus unidirectional.
- In Aggregation, both entries can survive individually, which means ending one entity will not affect another.

Example of aggregation in Python

```
class Salary:
  def __init__(self, pay, bonus):
    self.pay = pay
    self.bonus = bonus

  def annual_salary(self):
    return (self.pay*12)+self.bonus

class EmployeeOne:
  def __init__(self, name, age, sal):
    self.name = name
    self.age = age
    self.agg_salary = sal   # Aggregation

  def total_sal(self):
    return self.agg_salary.annual_salary()

  salary = Salary(10000, 1500)
  emp = EmployeeOne('Geek', 25, salary)
  print(emp.total_sal())
```

More relationships...

- `Association`: which expresses a uses-a relationship. For example, a student may be associated with a course. They will use the course. This relationship is common in database systems with one-to-one, one-to-many, and many-to-many associations.
- **Delegation**: which models a **can-do** relationship, where an object hands a task over to another object, which takes care of executing the task.
- **Dependency injection**: a design pattern you can use to achieve loose coupling between a class and its components. With this technique, you can provide an object's dependencies from the outside rather than inheriting or implementing them in the object itself.

Polymorphism

Polymorphism in Python

A set of classes implementing the same interface with specific behaviours for concrete classes is a great way to unlock polymorphism.

Polymorphism is when you can use objects of different classes interchangeably because they share a common interface.

Example

Python strings, lists, and tuples are all sequence data types. This means that they implement an interface that's common to all sequences.

We can use them in similar ways. For example, you can:

- Loop them because they provide the `.__iter__()` method
- Item access their through the `.__getitem__()` method
- Determine their number of items because they include the `.__len__()` method

Wrapping up the OOP in Python

1. Python classes and how to use them to make your code more reusable, modular, flexible, and maintainable
2. Classes are the building blocks of object-oriented programming in Python
3. With classes, you can solve complex problems by modelling real-world objects, their properties, and their behaviours
4. Classes provide an intuitive and human-friendly approach to complex programming problems, making your life more pleasant.
5. We can use special classes such as interfaces and abstract classes to unlock properties like polymorphism in python
6. Classes can interact through associations, aggregations, composition, inheritance, dependency injection and delegation