Classes and Objects

Assoc. Prof. Arthur Molnar

Babeș-Bolyai University

2024

Overview

1. Classes and Objects
   - Why define new types?
   - Classes
   - Objects
   - Class Methods, Fields
   - Special methods. Overloading
2. Python scope and namespace
   - Class vs instance attributes
3. Encapsulation. Information Hiding

Classes and Objects

NB!

Types classify values. A type denotes a **domain** (a set of values) and **operations** on those values.

Classes and Objects

Remember!

- **Procedural programming** - programs are assembled from a set of subroutines (or procedures, or functions) that talk to one another via input and return parameters.
- **Modular programming** - a software design technique that increases the extent to which software is composed of independent, interchangeable components called **modules**, each of which is responsible for one aspect within the program and contains everything necessary to accomplish this.

Object oriented programming

A programming paradigm that uses objects that have data and which "talk" to each other to build applications.

Why define new types?

Let's review the modular calculator example (lecture.examples.ex30_modular_calc):

1. Issues with global variables, if they exist:
   - You can easily break global variables by accessing them from different places.
   - They make testing difficult as their values cannot be easily controlled.
   - Managing the relation between them is difficult.
2. Issues without global variables:
   - The state of the calculator is exposed to the world.
   - The state has to be transmitted as parameter to every function .

Classes

What is a class

A construct used as a template to create instances of itself - referred to as class instances, class objects, instance objects or simply **objects**. A class defines constituent members which enable these class instances to have *state* and behaviour.

Classes in Python

- Defined using the keyword **class** (as in many other languages)
- The class definition is an executable statement.
- The statements inside a class definition are usually function definitions, but other statements are allowed
- When a class definition is created, a new namespace is created, and used as the local scope - thus, all assignments to local variables go into this new namespace (remember the **LEGB** rule from the modular programming section?). In particular, function definitions bind the name of the new function here.

Objects

What is an object

An **object** refers to a particular instance of a class, and is a combination of variables, functions and other data structures. Objects support two kinds of operations: **attribute (data or method) references** and **instantiation**.
Objects are useful because they allow us to ship state (attributes) and behaviour (methods) together, and control who and how can change and update them

Objects

1. Object instantiation - uses the reserved function notation of **__init__**
   - The instantiation operation creates an empty object that is of the type of the given class
   - A class may define a special method named **__init__**, used to create an instance of that class
   - In Python, use **self** to refer to that instance (in many other languages, it is the **this** keyword)

Objects

2. Attribute references (method or field)
   - Uses the "dot-notation", not dissimilar to **package.module** names.
   - We have instance variables/methods and class variables/methods
     - Instance variables/methods are specific to an object, with each object having its own instance of the variables/methods
     - Class variables/methods are specific to a class and are thus shared among all objects of the class
   - The variable/method referencing the object specifies on which instance the call is made, in the case of instance variables

Fields, Methods

- Fields (attributes)
  - Variables that store data specific to an instance or a class (see the slide above)
  - Can be objects themselves
  - They come into existence first time they are assigned to
- Methods
  - Functions in a class that can access values from a specific instance.
  - In Python the method will automatically receive a first argument: the current instance
  - All instance methods need to have the **self** argument

Class Methods, Fields

First example of using classes in Python

lecture.examples.ex32_python_class_particularities.py

Creating a new data type - Rational

lecture.examples.ex33_rational_number_basic.py

Special methods

- **__str__** - converts the current object into a string type (good for printing)
- **__eq__** - test (logical) equality of two objects
- **__ne__** - test (logical) inequality of two objects
- **__lt__** - test x $<$ y
- Many others at[1]

[1] https://docs.python.org/3/reference/datamodel.html

Special methods - operator overloading

- **__add__(self, other)** - to be able to use "+" operator
- **__mul__(self, other)** - to be able to use the "*" operator
- **__setItem__(self,index, value)** - to make a class behave like an array/dictionary, use the "[]"
- **__getItem__(self, index)**- to make a class behave like an array
- **__len__(self)** overload len
- **__getslice__(self,low,high)** - overload the slicing operator
- **__call__(self, arg)** - to make an object behave like a function, use the "()"

Special methods - example

Let's make our rational number type a bit more Pythonic and ... useful

lecture.examples.ex34_rational_number_operators.py

Python scope and namespace

NB!

- A *namespace* is a mapping from names to objects.
- Namespaces are implemented as Python dictionaries
  - Key: name
  - Value - Object
- Remember **globals()** and **locals()** ?

Python scope and namespace

- A class introduces a new namespace
- Methods and fields of a class are in a separate namespace (the namespace of the class)
- All the rules (bound a name, scope/visibility, formal/actual parameters, etc.) related to the names (function, variable) are the same for class methods and fields. Keep in mind that the class has its own namespace

Class vs instance attributes

- Instance attributes
  - The **self** reference decides for what object the attribute is accessed
  - Each instance has its own set of fields
- Class attributes
  - Attributes that are unique to the class
  - They are shared by all instances of the same class
  - In most languages, they are referred to as "static" fields, or methods
  - In Python, the **@staticmethod** decorator is used
  - Static methods do not receive the **self** reference

Instance vs. class fields

lecture.examples.ex35_instance_vs_class_fields.py

Class vs instance attributes

Discussion

Can you think of examples where class attributes are more suitable than instance attributes?

Encapsulation

- A set of rules or guidelines that you will use when deciding on the implementation of new data types
- What we will cover
  - Encapsulation
  - Information hiding
  - Abstract data types

Encapsulation

- The **state** of the object is the data that represents it (in most cases, the class attributes)
- The **behaviour** is represented by the class methods
- Encapsulation means that **state** and **behaviour** are kept together, in one **cohesive** unit

Information hiding

- The internal representation of an object needs to be hidden from view outside of the object's definition
- Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state
- Divide the code into a public interface, and a private implementation of that interface

Information hiding

- Define a specific interface and isolate the internals to keep other modules from doing anything incorrect to your data
- Limit the functions that are visible (part of the interface), so you are free to change the internal data without breaking the client code
- Write to the **Interface**, not the the **Implementation**
- If you are using only the public functions you can change large parts of your classes without affecting the rest of the program

Information hiding

Public and private members - data hiding in Python

- We need to protect (hide) the internal representation (the implementation)
- Provide accessors (getters) to the data
- Encapsulations is particularly important when the class is used by others

Information hiding

Public and private members - data hiding in Python

- Data hiding in Python is based upon convention
- Use **_name** or **__name** for fields, methods that are "private"
- A name (function, method, module-level variable or class field) prefixed with an underscore (e.g. **_spam**) should be treated as non-public. It should be considered an implementation detail and subject to change without notice.
- A name prefixed with two underscores (e.g. **__spam**) is private and name mangling is employed by Python

Guidelines

- Upper application layers do not have to know about implementation details of the methods or the internal data representation used by the code they call
- Code must work even when the implementation or data representation are changed
- Function and class specification have to be independent of the data representation and the method's implementation **(Data Abstraction)**

Abstract data types

- Operations are specified independently of their implementation
- Operations are specified independently of the data representation
- Abstract data type is a *Data type* + *Data Abstraction* + *Data Encapsulation*