Object-Oriented Programming in Python

Software Applications
A.Y. 2020/2021

OOP

- Object-oriented programming (OOP) is a programming language model in which programs are organised around data, or objects, rather than functions and logic.
- An object can be defined as a data field that has unique attributes and behaviour.
- Examples of an object can range from physical entities, such as a human being that is described by properties like name and address, down to small computer programs, such as widgets.
- This opposes the historical approach to programming where emphasis was placed on how the logic was written rather than how to define the data within the logic.

OOP in a nutshell

- The first step in OOP is to identify all of the ***objects*** a programmer wants to manipulate and how they relate to each other, an exercise often known as data modelling.
- Once an object is known, it is generalised as a ***class*** of objects that defines the kind of data it contains and any logic sequences that can manipulate it. Each distinct logic sequence is known as a method and objects can communicate with well-defined interfaces called messages.
- OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them. This approach to programming is well-suited for programs that are large, complex and actively updated or maintained.

Procedural programming Vs Object-Oriented Programming

- Procedural programming creates a step by step program that guides the application through a sequence of instructions. Each instruction is executed in order.
- Procedural programming also focuses on the idea that all algorithms are executed with functions and data that the programmer has access to and is able to change.
- Object-Oriented programming is much more similar to the way the real world works; it is analogous to the human brain. Each program is made up of many entities called objects.
- Instead, a message must be sent requesting the data, just like people must ask one another for information; we cannot see inside each other’s heads.

What is an Object?

- Objects are the basic run-time entities in an object-oriented system.
- They may represent a run-time instances of a person, a place, a bank account, a table of data or any item that the program must handle.
- When a program is executed the objects interact by sending messages to one another.
- Objects have two components:
  - Data (i.e., attributes)
  - Behaviours (i.e.,methods)

- An object, practically speaking, is a segment of memory (RAM) that references both data (referred to by instance variables) of various types and associated functions that can operate on the data.

What is a class?

- A class is a special data type which defines how to build a certain kind of object.
- The class also stores some data items that are shared by all the instances of this class
- Instances are objects that are created which follow the definition given inside of the class

- Functions belonging to classes are called methods. Said another way, a method is a function that is associated with a class, and an instance variable is a variable that belongs to a class.

Classes Vs Objects

- A class is a prototypical definition of entities of a certain type and it is defined by the programmer at coding time
    - Example - the class **Gene**
- Those entities are the objects that are created at runtime during the execution of the code
    - Example
        - **Gene AY342 with sequence CATTGAC**
        - **Gene G54B with sequence TTACTAGA**

OOP in Python

- Python is naturally “object oriented”
- In Python, objects are the data that we have been associating with variables.
    - Example: genes = ["AY342", "G54B"]
- What the methods are, how they work, and what the data are (e.g., a list of numbers, dictionary of strings, etc.) are defined by a class: the collection of code that serves as the “blueprint” for objects of that type and how they work.
    - Example:
        - the array class in Python defines a number of methods like “append”, “pop”, “sort”, etc.
        - genes.append("AY351")

OOP in Python (contd.)

- Thus the class (much like the blueprint for a house) defines the structure of objects, but each object's instance variables may refer to different data elements so long as they conform to the defined structure (much like how different families may live in houses built from the same blueprint).
- In Python, each piece of data we routinely encounter constitutes an object. Each data type we've dealt with so far (lists, strings, dictionaries, and so on) has a class definition—a blueprint—that defines it. For example, lists have data (numbers, strings, or any other type) and methods such as .sort() and .append().

Classes in Python

- Definitions for Python classes are just blocks of code, indicated by an additional level of indentation (like function blocks, if-statement blocks, and loop blocks). Each class definition requires three things:
  - methods (functions) that belong to objects of the class, e.g. *append(), sort()*
  - Instance variables referring to data, i.e. the attributes
  - A special method called a constructor. This method will be called automatically whenever a new object of the class is created, and must have the name __init__.

Example: the Gene class

Constructor

Attributes

Methods

```
#!/usr/bin/env python

class Gene:
    def __init__(self, creationid, creationseq):
        print("I'm a new Gene object!")
        print("My constructor got a param: " + str(creationid))
        print("Assing that param to my id instance variable...")
        self.id = creationid
        print("Similarly, assigning to my sequence instance variable...")
        self.sequence = creationseq

    def print_id(self):
        print("My id is: " + str(self.id))

    def print_sequence(self):
        print("My sequence is: " + str(self.sequence))


    def base_composition(self, base):
        base_count = 0
        for index in range(0, len(self.sequence)):
            base_i = self.sequence[index]
            if base_i == base:
                base_count = base_count + 1
        return base_count

    def gc_content(self):
        g_count = self.base_composition("G")
        c_count = self.base_composition("C")
        return (g_count + c_count)/float(len(self.sequence))

    def get_seq(self):
        return self.sequence

    def set_seq(self, newseq):
        assert self.base_composition("U") == 0, "Sorry, no RNA allowed."
        self.sequence = newseq
```

Class name

Instantiating Objects with ‘__init__’

- __init__ is the default constructor
- __init__ serves as a constructor for the class. Usually does some initialisation work
- An __init__ method can take any number of arguments
- However, the first argument self in the definition of __init__ is special

Self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument ***self***
- In __init__, self refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called
- Similar to the keyword this in Java or C++
- But Python uses self more often than Java uses this
- You do not give a value for this parameter(self) when you call the method, Python will provide it. Although you must specify self explicitly when defining the method, you don't include it when calling the method. Python passes it for you automatically

```
def print_sequence(self):
    print("My sequence is: " + str(self.sequence))
```

```
geneB.print_id()
```

Deleting instances: No Need to “free”

- When you are done with an object, you don’t have to delete or free it explicitly.
- Python has automatic garbage collection.
- Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
- Generally works well, few memory leaks
- There’s also no “destructor” method for classes

Fundamental concepts of OOP in Python

- The four major principles of object orientation are:
  - Encapsulation
  - Data Abstraction
  - Inheritance
  - Polymorphism

Encapsulation

- Important advantage of OOP consists in the encapsulation of data. We can say that object-oriented programming relies heavily on encapsulation.
- The terms encapsulation and abstraction (also data hiding) are often used as synonyms. They are nearly synonymous, i.e. abstraction is achieved though encapsulation.
- Data hiding and encapsulation are the same concept, so it's correct to use them as synonyms
- Generally speaking encapsulation is the mechanism for restricting the access to some of an objects's components, this means, that the internal representation of an object can't be seen from outside of the objects definition.

Encapsulation (contd.)

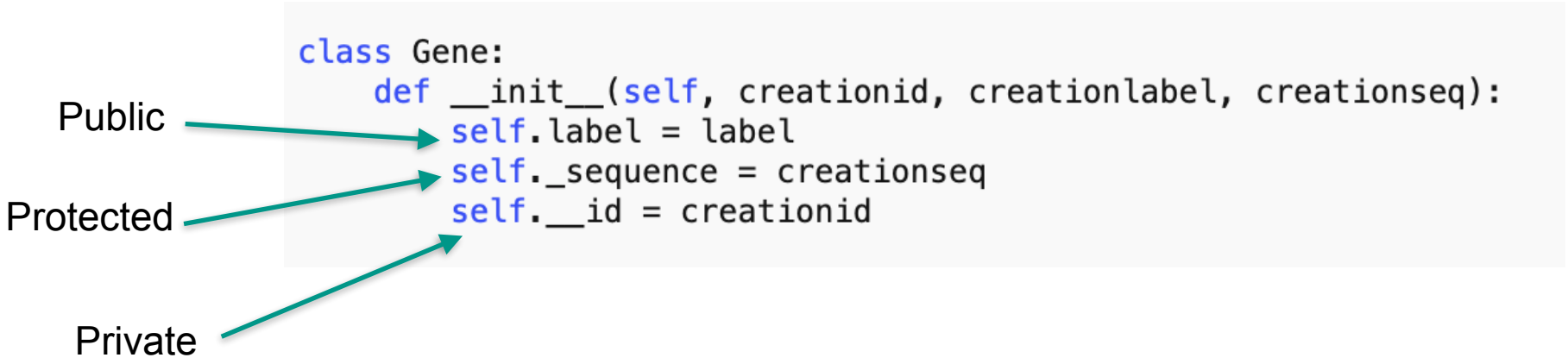
- Access to this data is typically only achieved through special methods: Getters and Setters
- By using solely get() and set() methods, we can make sure that the internal data cannot be accidentally set into an inconsistent or invalid state.
- C++, Java, and C# rely on the public, private, and protected keywords in order to implement variable scoping and encapsulation
- It's nearly always possible to circumvent this protection mechanism

Public, Protected and Private Data

- The following table shows the different behaviour Public, Protected and Private Data

Attribute Name	Notation	Behaviour
sequence	Public	Can be accessed from inside and outside
_sequence	Protected	Like a public member, but they shouldn't be directly accessed from outside
__sequence	Private	Can't be seen and accessed from outside

Example

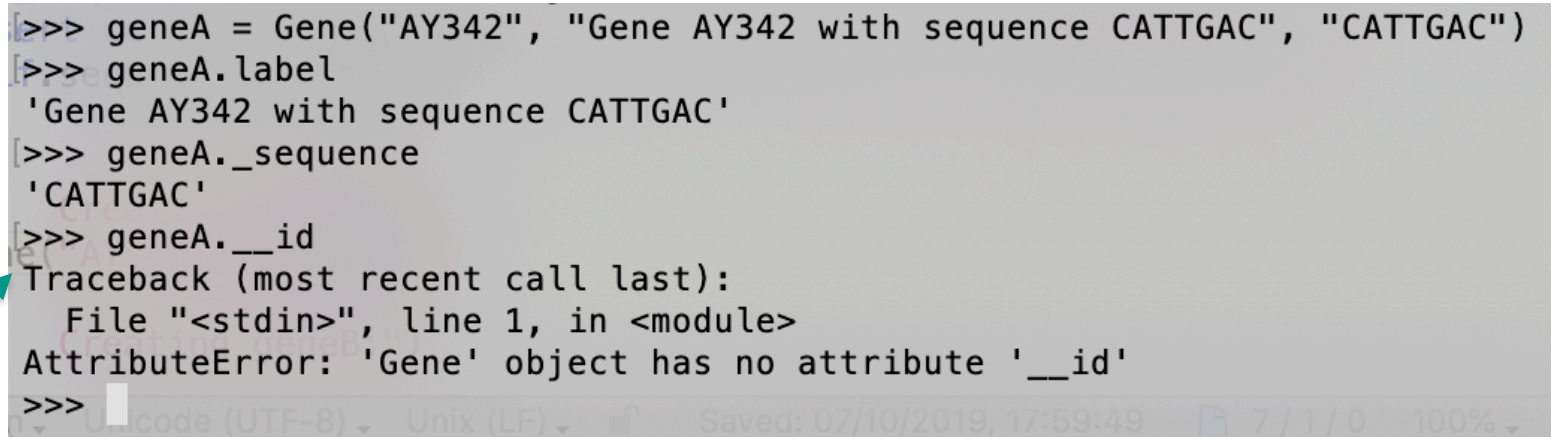

Public
Protected
Private
class Gene:
def __init__(self, creationid, creationlabel, creationseq):
self.label = label
self._sequence = creationseq
self.__id = creationid

Example (contd.)

OK!

OK, but it is a bad practice

Scoping error



```
>>> geneA = Gene("AY342", "Gene AY342 with sequence CATTGAC", "CATTGAC")
>>> geneA.label
'Gene AY342 with sequence CATTGAC'
>>> geneA._sequence
'CATTGAC'
>>> geneA.__id
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Gene' object has no attribute '__id'
>>>
```

How can we access and modify private attributes?

- We can access and modify private attributes with class ***getters*** and ***setters*** defined as class methods

Getter →

Setter →

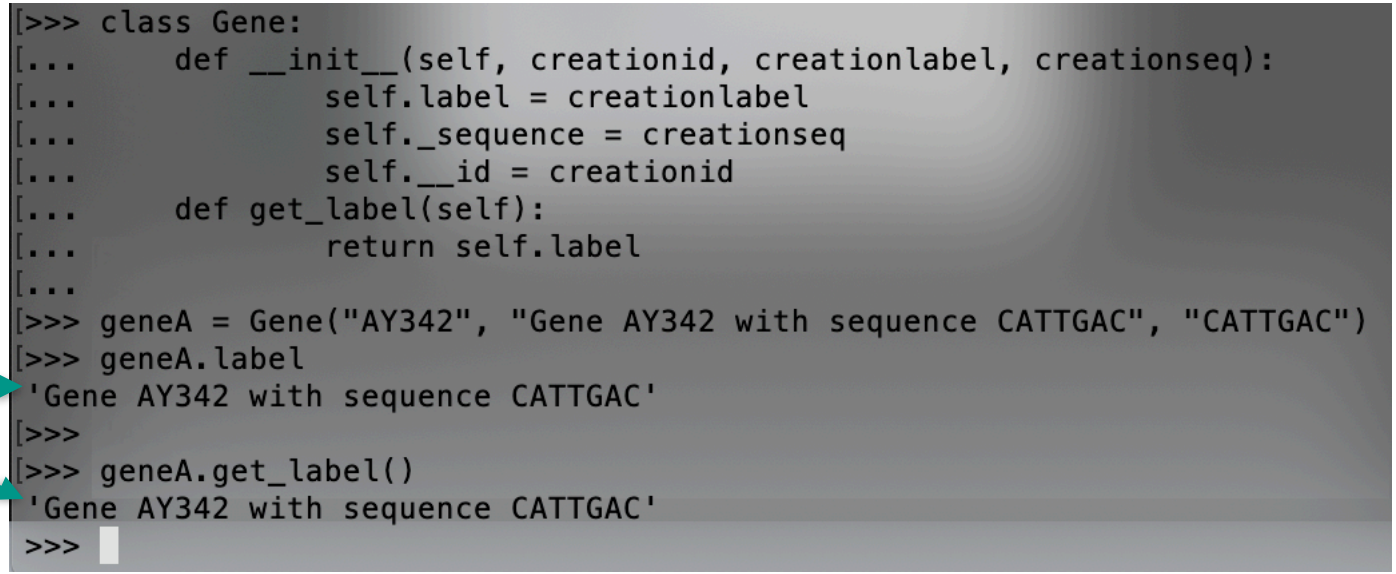
```
def get_id(self):
    return self.__id

def set_id(self, creationid):
    seld.__id = creationid
```

```
>>> geneA = Gene("AY342", "Gene AY342 with sequence CATTGAC", "CATTGAC")
>>> geneA.get_id()
'AY342'
>>> geneA.set_id("AY342_1")
>>> geneA.get_id()
'AY342_1'
>>>
```

Why should we use getters and setters for public attributes?

Same behaviour



```
>>> class Gene:
...     def __init__(self, creationid, creationlabel, creationseq):
...             self.label = creationlabel
...             self._sequence = creationseq
...             self.__id = creationid
...     def get_label(self):
...             return self.label
...
>>> geneA = Gene("AY342", "Gene AY342 with sequence CATTGAC", "CATTGAC")
>>> geneA.label
'Gene AY342 with sequence CATTGAC'
>>>
>>> geneA.get_label()
'Gene AY342 with sequence CATTGAC'
>>>
```

Why should we use getters and setters for public attributes?

- **Answer:** The difference is related a bit to politeness
    - By using methods, we are requesting that the object change its sequence data, whereas directly setting instance variables just reaches in and changes it—which is a bit like performing open-heart surgery without the patient’s permission!

Inheritance

- Inheritance is a powerful feature in object oriented programming
- It refers to defining a new class with little or no modification to an existing class.
- The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.
- Derived class inherits features from the base class, adding new features to it.
- This results into re-usability of code.

Example

- RegulatorGene is subclass of Gene

```
>>> class RegulatorGene(Gene):
...     def __init__(self, creationid, creationlabel, creationseq, controlled_genes):
...             super().__init__(creationid, creationlabel, creationseq)
...             self.controlled_genes = controlled_genes
...     def get_controlled_genes(self):
...             return self.controlled_genes
...
>>> geneB = RegulatorGene("AY345", "Gene AY345 with sequence CAATGTC", "CAATGTC", "AY342")
>>> geneB.get_controlled_genes()
'AY342'
>>>
```

Coding exercise: data modelling

- Code a class that represents a binary tree, which is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. Each node is a class that contains a value.
- The class for a binary tree provides methods for:
  - computing the height of the tree
  - counting the number of nodes
  - adding new nodes by automatically checking where it is possible to add a new node in the binary tree
  - removing nodes
- The class for a node provides methods for:
  - getting/setting/removing a value for the node
  - getting/setting/removing left/right child
  - checking if the left/right child is available

Coding exercise: instantiation