Lecture 7 (II)
Object-Oriented Programming (OOP)

GNBF5010

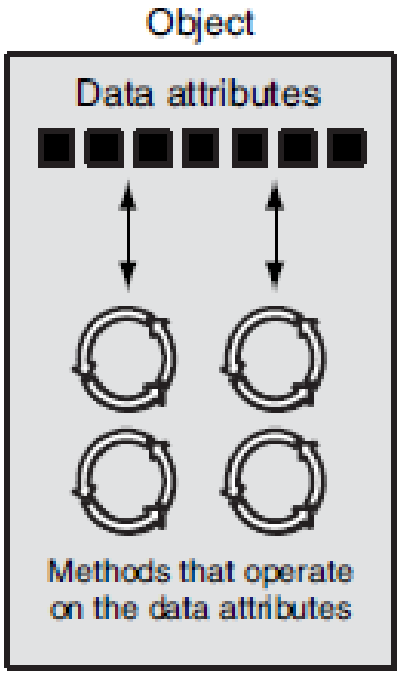
Instructor: Jessie Y. Huang

Overview

- Main idea of OOP design
- How to create a class and the objects of it
- More about class methods

What is OOP?

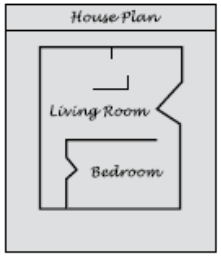
- **Object-oriented programming (OOP)** is centered on creating ***objects***, which are entities that contain both data and procedures.
  - Data contained in the object are known as ***attributes***.
  - The procedures that an object performs are known as ***methods***.
  - The methods of an object perform operations on the data attributes.



Class and object

- A ***class*** is the code that specifies the *data attributes* and *methods* for a particular type of ***object***.
  - Think of a class as a "blueprint" that must be created prior to the creation of objects.

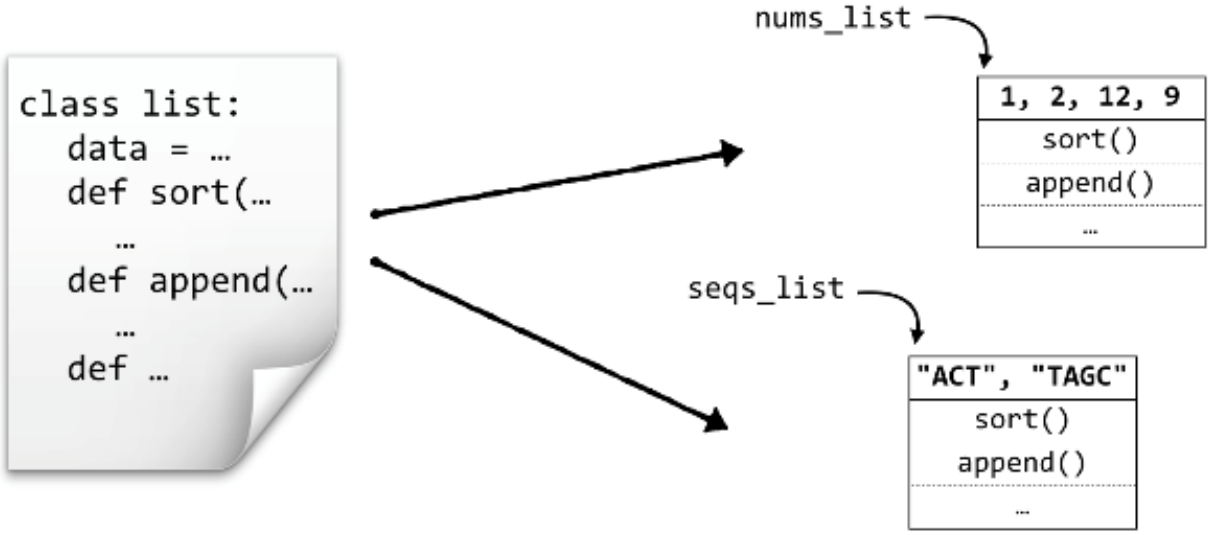
Blueprint that describes a house



Instances of the house described by the blueprint

Objects in Python

- Each object we've dealt with so far (lists, strings, dictionaries, and so on) has a class definition.
  - Lists have data (numbers, strings, or any other type) and methods such as .sort() and .append().



Class definition: code written that defines the structure of objects

Objects: data and functions in RAM, referenced by variables

Objects in Python

- Everything in Python is an object

```
n = 12              # n is an object of type integer
s = 'ACAGATC'       # s is an object of type string
l = [12, 'A', 21121]   # l is an object of type list
```

- Objects contain data (the number 12, the string 'ACAGCTC', …)
- Objects can be modified/manipulated by calling their methods

```
s.count('A')
s.lower()
l.append('121')
```

It might be useful to create **your own data types** and objects
that built to your own specifications and organized in the way convenient to you.

How to create a new class

Creating a class

- To create a class, use the keyword **class**
- Example:
  Create a class named MyClass, with an attribute named x:

```
class MyClass:
    x = 5
```

A simple class, not useful in real life.

  Create a MyClass object and access the attribute x:

```
p1 = MyClass()
print(p1.x)
```

The constructor function __init__()

- Each class should have a constructor function named `__init__()`
  - Called when a new object of the class is created
  - Used to assign values to the object attributes
  - Provides a quick visual reference when coding
- Example: Create a class named Person, use the **`__init__()`** function to assign values for name and age.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

The object methods

- Methods are functions that belong to an object.
- A method must take the **`self`** parameter as its first argument.
- Example: Create a method in the Person class which prints a greeting; then execute it on the p1 object:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greeting(self):
        print("Hello! My name is " + self.name)

p1 = Person("John", 36)
p1.greeting()
```

The mandatory ***self*** parameter references to the current object of the class.

Parameters specified upon object creation will be passed to the constructor function

Creating a class: A complete example

Keyword

Class name

Default arguments

Constructor function

Attributes

Class methods

```
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')

# Create a ComplexNumber object and call get_data()
num1 = ComplexNumber(2, 3)
num1.get_data()                          # Output: 2+3j

# Create another ComplexNumber object and print data attributes
num2 = ComplexNumber(5)
print((num2.real, num2.imag))            # Output: (5, 0)
```

Three steps for a class definition

1. **Decide** what entity the objects will represent, as well as what **data** (attribute variables) and **methods** (functions) they will have.
2. Create a **constructor** method __init__() and use it to initialize all **attribute** variables,
   - either with parameters passed into the constructor function,
   - or with default values if you don't want to initialize the attributes upon object creation.
3. Write **methods**
   - e.g. set or get the attribute variables, compute calculations, call other methods or functions.
   - Don't forget the **self** parameter!

Example: The Gene Class

- Let's define a class called Gene:
  - Each Gene object will have an id and a sequence.
  - Upon object creation, id and sequence will be passed to __init__()
- Outside the class definition block, we can use the Gene class to create Gene objects.

Example: The Gene Class

```
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 attribute variable...")
        self.id = creationid
        print("Similarly, assigning to my sequence attribute 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))


print("***   Creating geneA:")
geneA = Gene("AY342", "CATTGAC")

print("***   Creating geneB:")
geneB = Gene("G54B", "TTACTAGA")

print("***   Asking geneA to print_id():")
geneA.print_id()

print("***   Asking geneB to print_id():")
geneB.print_id()

print("***   Asking geneA to print_sequence():")
geneA.print_sequence()
```

Example: The Gene Class

Resulting in the output:

```
***   Creating geneA:
I'm a new Gene object!
My constructor got a param: AY342
Assigning that param to my id attribute variable...
Similarly, assigning to my sequence attribute variable...

***   Creating geneB:
I'm a new Gene object!
My constructor got a param: G54B
Assigning that param to my id instance variable...
Similarly, assigning to my sequence instance variable...

***   Asking geneA to print_id():
My id is: AY342

***   Asking geneB to print_id():
My id is: G54B

***   Asking geneA to print_len():
My sequence len is: 7
```

Advantages of OOP

- Bundle together objects that share
  - common attributes and
  - procedures that operate on those attributes

- Classes make it easy to reuse code:
  - many Python modules define new classes
  - each class has a separate environment (no collision on function names)
  - inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

More about class methods

- A method can call another method of the same object.

```
    # ... (inside class Gene:)

    def print_len(self):
        print("My sequence len is: " + str(len(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))

print("\n***   Creating geneA:")
geneA = Gene("AY342", "CATTGAC")

# ...

print("\n***   Asking geneA to return its T content:")
geneA_t = geneA.base_composition("T")
print(geneA_t)


print("\n***   Asking geneA to return its GC content:")
geneA_gc = geneA.gc_content()
print(geneA_gc)
```

gc_content() calls base_composition() function inside a Gene object.

```
***   Asking geneA to return its T content:
2

***   Asking geneA to return its GC content:
0.428571428571
```

The getter and setter methods

- Define “getter” and “setter” methods to get and set attributes outside the class. (Good practice)
  - Good to hide attributes; easy to maintain code; easy to debug

```
    # ... (inside class Gene:)

    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):
        if newseq.base_composition("U") != 0:
            print("Sorry, no RNA allowed.")
        else:
            self.sequence = newseq


# ...
print("gene A's sequence is " + geneA.get_seq())
geneA.set_seq("ACTAGGGG")
```

For example, in setter method, validate the data before accepting it.

Use getter and setter outside the class, instead of geneA.sequence.

Customizing object comparison using class methods

- Objects can be compared with ==, <, and other operators, if some special methods are implemented in class.
- We might say geneA and geneB are equal if their sequences are the same, and geneA is less than geneB if geneA.seq < geneB.seq.
- Thus, we can add a special method `__eq__()`, which
  - takes another object (of the same type) as an argument and
  - returns **True** if the two **equal** and **False** otherwise

```
def __eq__(self, other):
    if self.seq == other.get_seq():
        return True
    return False
```

- We can also implement an `__lt__()` method for “less than”:

```
def __lt__(self, other):
    if self.seq < other.get_seq():
        return True
    return False
```

Customizing object comparison using class methods

- Similarly, other comparisons can be enabled by defining
  - `__le__()` for less or equal, <=
  - `__gt__()` for greater than, >
  - `__ge__()` for greater or equal, >=
  - `__ne__()` for not equal, !=

```
geneA = Gene("XY6B", "CATGATA")
geneB = Gene("LQQ5", "CATGATA")

print(geneA.__lt__(geneB))  # False
# same as:
print(geneA < geneB)        # False

print(geneA.__eq__(geneB))  # True
# same as:
print(geneA == geneB)       # True
```

Exercise

1. Now modify the **gene_class.py** program and implement the `__eq__()`, `__lt__()`, and the other comparison methods mentioned above. Then compare two objects of genes with the standard comparison operators see if they works as expected.

2. If we have a list of Gene objects **genes_list**, where the comparators mentioned above are defined, then Python can sort according to our comparison criteria with **genes_list.sort()** and **sorted(genes_list)**. Now create a list of objects of the class and sort the list.

3. Implement a `__repr__()` method in the class, which should return a nicely formatted string that represents the object. In fact, the `__repr__()` method will enable the print() method of this object.

Reading

- Chapter 23, Part II, A Primer for Computational Biology
- Chapter 10 & 11, Starting out with Python