

Topic 9

Object-Oriented Programming



Murdoch
UNIVERSITY

Last Topic

- Testing
- Debugging
- Catch exceptions
- Raise exceptions
- Assertions
- Defensive programming

This Topic

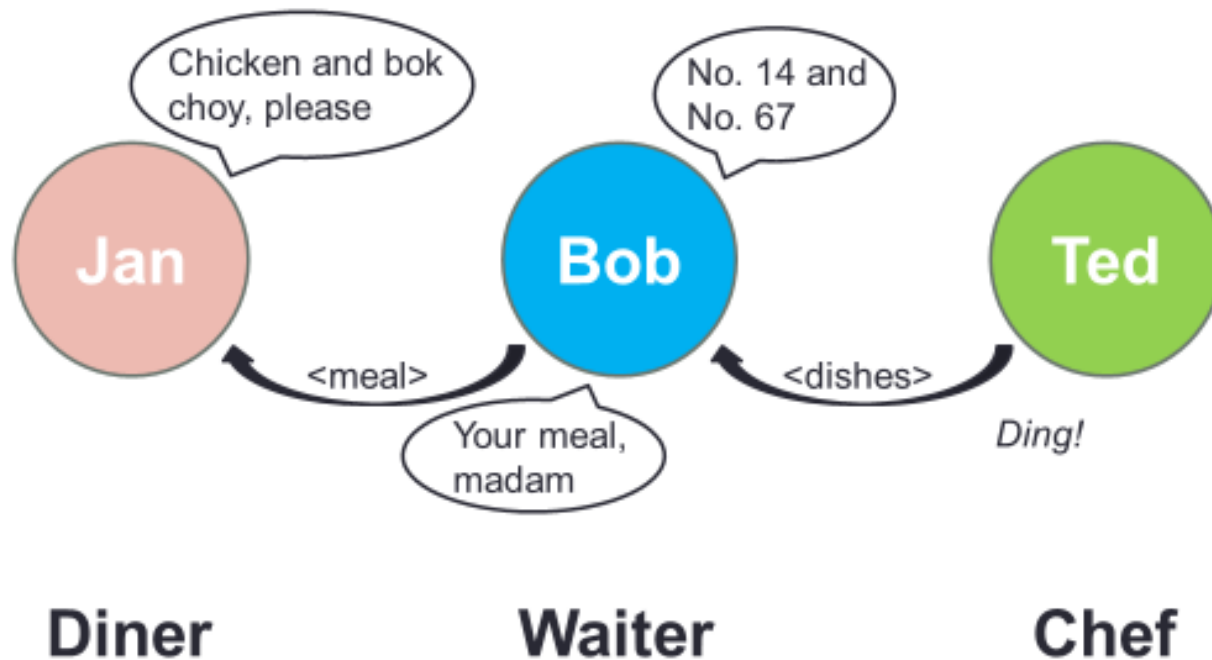
- Object-oriented programming
- Object-oriented programming languages
- Classes and objects
- Class definition
- Class members vs instance members
- Create objects from a class
- Operator methods
- Solving problems with classes and objects

Object-Oriented Programming (OOP)

- You must have heard the term OOP a lot, but what exactly does it mean?
- OOP refers to a style of programming which is different from the earlier dominant programming methodology known as *procedural programming*.
- With procedural programming, one focuses on dividing the task into a sequence of steps and this process becomes the basis for the program design.
- In contrast, OOP starts by analysing the task and identifying the objects in the task. The program design is centred on the interactions between these objects.

Object-Oriented Programming (OOP)

Object Orientation



Object-Oriented Programming (OOP)

- There is often a misconception: to write an object-oriented program, you must use an object-oriented programming language.
- This is not the case. You can write an object-oriented program using a non-object-oriented programming language such as C. However, it is not as convenient as using an object-oriented programming language.
- Most old programming languages such as C, COBOL, FORTRAN are non-object-oriented programming languages.
- Most new programming languages are OO languages, eg, C++, C#, Java, Python, JavaScript.

Object-Oriented Programming Language

- What is an OO language? An OO language must support the following concepts:
 - **Encapsulation**: ability to bundle data members (state) and the methods (behaviours) that operate on that data into a single unit called an object. It also allows you to hide the internal details of how an object works, providing an interface for interacting with it. This is achieved by class and objects.
 - **Inheritance**: is a mechanism that allows you to create a new class (subclass or derived class) based on an existing class (base class or superclass). The subclass inherits data members and methods from the superclass and can also add its own data members and methods.
 - **Polymorphism**: the term means "many forms". In the context of OO programming, it means that objects of different classes can be treated as objects of a common superclass. It also means many related methods can have the same name. Polymorphism allows you to write more generic code that can work with objects of multiple related classes.

Classes and Objects

- An object is a computer representation of a real-world object, such as a car, a person, a course, etc. An object has a state (a set of data values) and behaviours (a set of methods that operate on the data).
- A class is a language construct representing one type of objects. The class serves as a template for the type of objects.
- The relationship between a class and its objects is akin to the relationship between a blueprint and the houses built with the blueprint.

Classes and Objects


Blueprint vs Houses	Class vs Objects
Using one blueprint, you can build multiple houses.	Using one class, you can create multiple objects
Although these houses differ from each other, they all share the same fundamental structure.	Although these objects differ from each other, they all share the same fundamental structure.
The blueprint doesn't require any physical resources such as brick and tiles, but each house must be built with physical resources.	The class doesn't require any physical resources such as memory, but each object must be stored in a piece of memory space.

Object Example: the list


[1,2,3,4]

- How is the list represented internally?
 - it is possible to place the four elements one after the other, compact but not good for adding and removing elements
 - or use a linked list shown below, good for adding and removing elements, but need to follow the links to find an element
- L** =


1		->
---	--	----



2		->
---	--	----



3		->
---	--	----



4		->
---	--	----
- the key is that these implementation details are kept private, the users of the lists do not need to know the details to use the lists – abstraction!
- The list has a set of data attributes, ie, 1, 2, 3, and 4
- It also has a set of operations:
 - List concatenation operator +
 - A set of methods, eg, `L.append()`, `L.extend()`, `L.count()`, `L.index()`, `L.insert()`, `L.pop()`, `L.remove()`, `L.reverse()`, `L.sort()`

Data Types and Classes

- Python supports many different *types* of data

```
121      3.14159    "Hello"   True   None
[ 1, 5, 7, 11, 13 ]
( 2, 7, 8)
{ 1, 3, 5, 7, 9 }
{ "WA": "Western Australia", "VIC": "Victoria" }
```

- In Python, each data type has a corresponding built-in class. Therefore, every data value is an object of some class in Python.

data value	class	data value	class
121	int	[1,5,7,11,13]	list
3.13159	float	(2,7,8)	tuple
"Hello"	str	{1,3,5,7,9}	set
True	bool	range(10)	range
None	NoneType	{"WA": "Western Australia" "VIC": "Victoria" }	dict

Advantage of OOP

- Bundle data into packages together with the methods that work on them through well-defined interfaces
- Divide-and-conquer:
 - implement and test the behaviour of each class separately
 - increased modularity reduces the complexity
- 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' behaviour

Define Our Own Class

- We have used many objects of some built-in classes in the past, but we haven't defined a class of our own.

- Syntax of a class definition:

```
class class_name:  
    class_variable  
    class_method  
    instance_method
```

- Under the reserved word `class`, you may define any number of class variables, class methods and instance methods and in any order.
- In the next example, we will briefly mention class variables and class methods, but our focus will be on instance methods and instance variables.
- An object of a class is also called an instance of that class.

Example

```
class Student:
    uni = "MU"                                # class variable

    @classmethod                               # decorator used to define a class method
    def setUni(cls, uni):                     # a class method only operates on class variables.
        cls.uni = uni                        # it should not operate on instance variables.

    def __init__(self, name, age):            # constructor, which is called automatically
        self.name = name                    # when a class is instantiated to set
        self.age = age                      # the instance variables
    def changeName(self, name):               # instance method
        self.name = name                   # self.name is an instance variable
    def changeAge(self, age):                 # instance method
        self.age = age                    # self.age is an instance variable

s1 = Student("John", 25)                     # create an instance of Student s1
s2 = Student("Mary", 23)                     # create another instance of Student s2

s1.setUni("UWA")                             # call the class method to change a class variable
print(s1.uni)                                # both print statements would print the same
print(s2.uni)                                # value "UWA"

s1.changeName("Peter")                       # call an instance method to change an instance
print(s1.name)                               # variable. This print will print "Peter"
print(s2.name)                               # no change, still print "Mary".

print(Student.uni)                           # class variable can be accessed using class name
print(Student.name)                          # but not instance variable: AttributeError
```

Class Variables and Class Methods

- A class member (class variable or class method) belongs to the whole class – all objects of the class share it.
 - It can be accessed via the object reference or the class name.
 - Change to a class variable via the class name applies to all objects of the class. Eg, `Student.uni = "Curtin"`
 - after the change, both `s1.uni` and `s2.uni` are "Curtin"
 - Change to a class variable via an object reference turns the class variable to an instance variable. Eg, `s1.uni = "Curtin"`
 - after the change, `uni` in `s1` becomes an instance variable with value "Curtin", but `s2.uni` is still a class variable with its old value.
 - not recommended.
 - A class method must be defined with `@classmethod` decorator
 - The first formal parameter of a class method always represents the class name, although it doesn't have to be named `cls`.

Instance Variables and Instance Methods

- An instance member (instance variable or instance method) belongs to the instance.
 - It can be accessed via the object reference only.
 - An instance variable can only be created, changed, or accessed via the object reference of the instance.
 - Changing an instance variable only affects one instance. Eg, `s1.name = "Hong"` only changes `name` in `s1` to "Hong"
 - The first formal parameter of an instance method always represents the object reference of the current object, although it doesn't have to be named `self`.
 - When defining a class, we usually defines an object constructor `__init__`, which is automatically called when an object is created to set the *instance variables*.

Creating and Using Your Own Classes

- Make a distinction between creating a class and using an instance of the class
- Creating the class involves
 - defining the class name
 - defining class attributes:
 - class variables
 - class methods
 - instance variables
 - instance methods
- Using the class involves
 - creating new instances of the class
 - doing operations on the instances
 - for example, `L=[1,2]` and `len(L)`, `L.append(4)`

Constructors of the Class

- first have to define how to create an instance of object, using defining a constructor method `__init__`.
- The constructor creates the instance variables for the object at the time the object is created

- Example

```
class Coordinate:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
    .....
```

- In the above example, the constructor creates two instance variables, `x` and `y` (**which belong to the whole object**) and initialise them to formal parameters `x` and `y` (which exist only within method `__init__`) respectively.

Creating an Instance of the Class

```
point = Coordinate(3,4)
origin = Coordinate(0,0)
print(point.x)
print(origin.x)
```

- Do not provide argument for self.
- Once the object is created, Python automatically calls the constructor by passing **the object reference of the new object** as the first argument and the two user specified arguments as the 2nd and 3rd arguments.
- Use operator "." to access the instance variable `x` as in `point.x` and `origin.x`

Defining an Instance Method

- A method in a class is like a function, but it works only within this class
- Python always passes the object reference as the first argument
- the convention is to use `self` as the name of the first formal parameter of a method, although you can use any name

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq+y_diff_sq)**0.5
```

How to Use the Method

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq+y_diff_sq)**0.5

p1 = Coordinate(3, 4)
p2 = Coordinator(5, 8)
distance = p1.distance(p2)
print(distance)
```

Print the Object

```
>>> point = Coordinate(3,4)
>>> print(point)
<__main__.Coordinate object at 0x7fa918510488>
```

- uninformative print representation by default
- define a `__str__` method for a class
- Python calls the `__str__` method when used with `print` on your class object
- you choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(point)
<3,4>
```

Defining Your Own __str__ Method

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return "<" + str(self.x) + ", " + str(self.y) + ">"
```

Wrapping Your Head Around Types and Classes

- can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
```

```
>>> print(c)
```

```
<3,4>
```

```
>>> print(type(c))
```

```
<class '__main__.Coordinate'>
```

return of the `__str__` method
the type of object `c` is a class `Coordinate`

- this makes sense since

```
>>> print(Coordinate)
```

```
<class '__main__.Coordinate'>
```

```
>>> print(type(Coordinate))
```

```
<type 'type'>
```

a `Coordinate` is a class
a `Coordinate` class is a type of object

- use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
```

```
True
```


Special Operators

- `+`, `-`, `==`, `<`, `>`, `len()`, `print`, and many others
- <https://docs.python.org/3/reference/datamodel.html#basic-customization>
- like `print`, can override these to work with your class
- define them with double underscores before/after

<code>__add__(self, other)</code>	→	<code>self + other</code>
<code>__sub__(self, other)</code>	→	<code>self - other</code>
<code>__eq__(self, other)</code>	→	<code>self == other</code>
<code>__lt__(self, other)</code>	→	<code>self < other</code>
<code>__len__(self)</code>	→	<code>len(self)</code>
<code>__str__(self)</code>	→	<code>print self</code>

... and others

Example: Fractions

- create a new type to represent a number as a fraction
 - because numbers such as $\frac{2}{3}$ do not have an exact representation as a float
 - $\frac{2}{3} = 0.6666666666666666666666666666\ldots$
- internal representation is two integers
 - numerator
 - denominator
- interface a.k.a. methods a.k.a how to interact with Fraction objects
 - add, subtract
 - print representation, convert to a float
 - invert the fraction

Example: Fractions

```
class Fraction:
    def __init__(self, n, d):
        self.numerator = n
        if d==0:
            raise Exception("Denominator cannot be 0")
        else:
            self.denominator = d

    def __str__(self):
        return str(self.numerator)+"/"+str(self.denominator)

    def __add__(self, other):
        f = Fraction(0,1)
        f.numerator = self.numerator * other.denominator +
other.numerator * self.denominator
        f.denominator = self.denominator * other.denominator
        return f
```

Example: Fractions

```
def __sub__(self, other):
    f = Fraction(0,1)
    f.numerator = self.numerator * other.denominator -
other.numerator * self.denominator
    f.denominator = self.denominator * other.denominator
    return f

def __eq__(self, other):
    if self.numerator/self.denominator ==
other.numerator/other.denominator:
        return True
    else:
        return False

f1 = Fraction(1, 3)
f2 = Fraction(3, 5)
f3 = Fraction(2, 6)
print(f1+f2)
print(f1-f2)
print(f1==f2)
print(f1==f3)
```

Summary

- bundle together objects that share
 - common attributes and
 - methods that operate on those attributes
- use abstraction to make a distinction between how to implement an object vs how to use the object
- build layers of object abstractions that inherit behaviors from other classes of objects
- create our own classes of objects on top of Python's basic classes