Classes and Objects

Abstraction

Abstraction means hiding the details about how an object performs its tasks.

We know what we can do with an object, but not how the object will do it. For example, we know that the list's index function will search a list, but we do not know how that algorithm is implemented.

We say that objects are a data abstraction that captures:

1. an internal representation (what we know about an object)
   - through data attributes
2. an interface for interacting with object (how we can use the object)
   - through methods (procedures/functions)
   - defines behaviors but hides implementation

Advantages of Object-Oriented Programming

With OOP we can store data together with the functions used to manipulate the data.

Divide-and-conquer development

- implement and test behavior of each class separately
- increased modularity reduces 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' behavior

Objects

Everything in Python is an object

Python supports many kinds of data. Some examples:

```
1234      3.14159      "Hello"      [1, 5, 7, 11, 13]
{"CA": "California", "MA": "Massachusetts"}
```

Each is an object, and every object has:

- a type
- an internal data representation (primitive or composite)
- a set of procedures for interaction with the object

An object is an instance of a type.

Classes

A class is a `type` which describes the design (data and behaviours) of a set of objects. For example, the str class describes the behaviors of all strings.

The class implements **methods** to define the behaviors of the object and **variables** to define the state of an object.

Once the type is defined, we can create (instantiate) many object of the same type to access their functionality.

Attributes – data and methods

Attributes are data and methods (functions) that are defined by the class.

Data attributes:

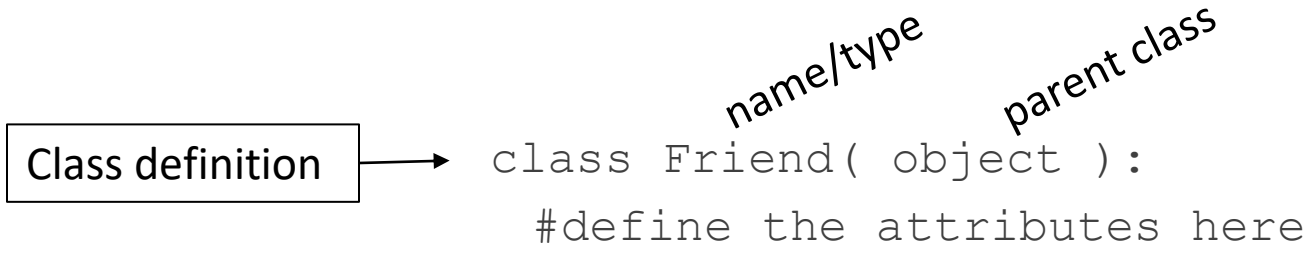
- Information about an object.
- For example, students have names, id numbers, courses.

Methods:

- Functions that only work with objects of the class.
- Interface to the class, how we interact with the object.
- What we can do with objects, for example students can calculate gpa.

Define your own Type (Classes)

Use the **`class`** keyword to define a new type.



Like in functions, indent code to indicate which statements are part of the class.

The word `object` means that the class is a Python object and inherits all its attributes (inheritance)

Creating instances: `__init__` method

Objects have attributes (store data), which are initialized when an object is created.

Every class has a special method `__init__` that is called when an object is created.

The `__init__` method is responsible for creating the object and initializing its attributes.

```
class Friend( object ):
  def __init__( self, name, phone, year ):
        self.fname = name
        self.fphone = phone
        self.year = year
```

Data attributes of the object and their values

Refers to the object being created

See: Friend_1.py

`self` Reference

Self refers to the current instance of the class and allows us to update/access the data of the current object.

When the `__init__` method is called, the first parameter is always the self reference, which is a reference to the object being initialized.

When we invoke a method on an object, for example `s1.index('abc')`, self refers to the object the method was invoked on, in this case `s1`.

Any method that uses object specific data has the `self` reference as the first parameter.

Instantiating Objects

Attributes of an object are also called instance variables, variables associated with a specific *instance* of a class (type).

When we create an object, we set the starting values of the instance variables.

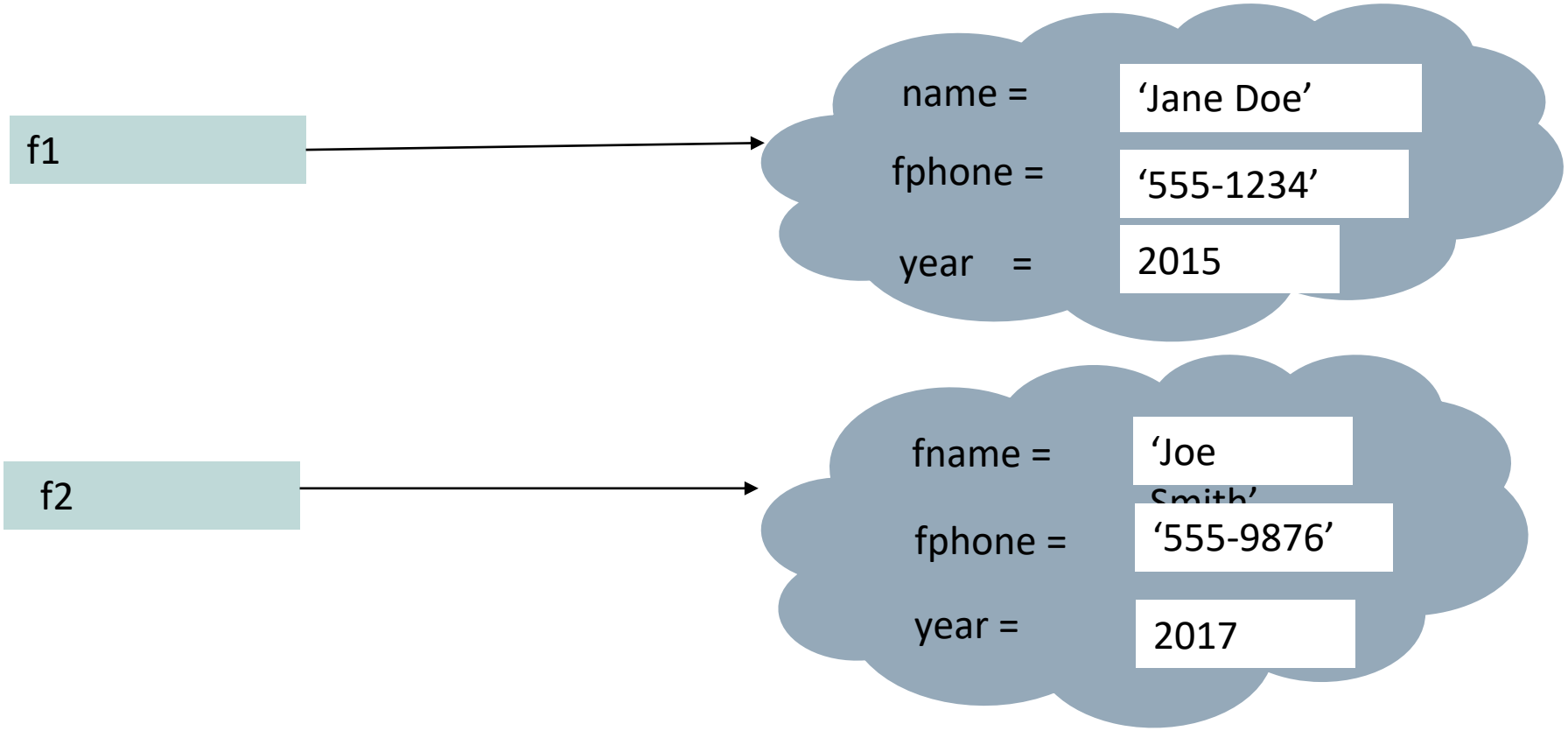
```
f1 = Friend('Jane Doe','555-1234',2015)
f2 = Friend('John Smith','555-9876',2017)
```

The statements above create 2 Friend objects.

The `__init__()` method is called implicitly (automatically) and sets the name, phone and year values of each Friend object using the values passed as parameters.

Notice that `self` is not passed as a parameter. Python automatically passes the `self` reference to the object's methods.

Instance Data



What is a Method

A function defined in a class, that uses object specific data.

As we have seen with other types of objects, we can access the methods of an object using the dot operator (.)

When a method is invoked on an object, the method invoked is automatically passed (`self`), and the method can access that object's data.

Defining Methods in Python

Our classes will define several methods. These methods will use data related to specific instances or objects.

Implementing a method is very similar to implementing a function, with one essential difference: you access the instance variables of the `self` object in the method body.

Every method must include the special `self` parameter variable, and it must be listed first. This variable refers to the object whose data will be used in the method.

Methods may take one or more parameters, but `self` must be the first.

Special Methods - string representation (__str__)

When we want to display an object or to concatenate an object with a string, we need to get the string representation of the object.

For example, when we print an object, python automatically converts the object to a string.

To convert an object to a string, the `str()` method is called.

By default, for our custom objects the str() method returns the address and type of the object, but not the object specific data.

If we print an object using the print statement, the address of the object will be displayed.

To access/return a customized string representation of our objects, we should define our own method that will be used to return the string representation of our object.

Our method replaces (overrides) the default method.

See: Friend_2.py

Special Methods - string representation (__repr__)

The `__str__()` and `__repr__()` methods are both used to get a String representation of an object.
The print statement and `str()` built-in function invokes `__str__` and the `repr()` built-in function invokes `__repr__` to display the object.

What is the difference?
`str()`: returns a user-friendly version of an object.
`repr()`: returns a developer friendly version of an object.

Example:
Create a datetime object and inspect its string representations.

```
import datetime
today = datetime.datetime.now()
```

```
print(str(today))  -> '2021-09-15 09:26:40.073045'
print(repr(today)) -> 'datetime.datetime(2021,9,15,9,26,40,73045)'
```

See: Friend_2.py

Encapsulation and Information Hiding

Two important principles of object-oriented programming.

Encapsulation: the bundling together of data attributes and the methods for operating on them.

Encapsulating data and behaviors in a class allows us to enable **information hiding** within a class.

Information hiding: The process of providing a public interface, while hiding the implementation details.

Private Attributes of Objects

To control how data of an object is accessed and updated, instance variables should be made private.

The convention for making an attribute private is to name with a leading ***double*** underscore (`__varName`)

Using the private naming convention, users cannot access private data directly from outside the class but instead access using the public interface (methods in the class).

Getter and Setter Methods

Each class defines an interface that allows public access to private data members (**data encapsulation**)..

Designers of a class decide which access to give for private data.

Instead of accessing an object's data members directly from outside the class, access is provided via:

- **get methods:** get the values of private data members.
- **set methods:** to update the values of private data members.

Objects can be made mutable/immutable via get and set methods.

`See: Friend_3.py`

Defining Methods – Friend Class

The reason we create objects is to define functionality to be used in applications.

As we have discussed, methods are functions defined in a class which use object specific data.

```
class Friend( object ):
  …
  def find_years( self ):
     print('You have known',self.__fname,'for', 2021 - self.__year,'years')
```

See: Friend_4.py

Car Example

Create a class Car with the following attributes:

- plateNo
- Brand
- Model
- Year
- Price

Define the following methods:

- init()
- Methods that return and update the values of each attribute
- repr/str methods

Write a script that creates 4 Cars, stores them in a list and does the following:

- Display the list of cars
- Input a plate number and display matching car, error if car does not exist
- Displays all cars produces after 2014
- Displays the average price of all cars in the list.

See: `Car.py / CarApplication.py`

Built in Functionality and Special Methods

What happens if we sort the list of Cars using the built-in sort method?

When we sort objects of a given type, the sort order depends on the object's attributes.

Often the sort order is application specific, how do we want to sort objects of a given type?

Let's assume cars should be sorted according to plateNo. We need to implement built-in special methods to define our own custom sort behavior.

The `list.sort()` function compares list objects using their built-in `__lt__` method.

For cars, we will implement `__lt__` to define a car being less than another based on their plateNo.

Special Methods

We can define and implement methods that will be called automatically when a standard Python operator (`+`, `-`, `*`, `/`, `==`, `<`, `>`…) is applied to an instance of the class.

This allows for a more natural use of the objects than calling the methods by name.

Example: To test whether two objects are equal, we could implement a method `isequal()` and use it as follows:

```
if obj1.isequal(obj2)
```

Instead of this, we can use the `==` operator and this is achieved by defining the special method `__eq__` which is called automatically when we compare two objects using the `==` operator.

Common Special Methods

Expression	Method Name
`x + y`	`__add__(self,y)`
`x - y`	`__sub__(self,y)`
`x * y`	`__mul__(self,y)`
`x / y`	`__truediv__(self,y)`
`x // y`	`__floordiv__(self,y)`
`x % y`	`__mod__(self,y)`
`x ** y`	`__pow__(self,y)`
`x == y`	`__eq__(self,y)`
`x != y`	`__ne__(self,y)`
`x < y`	`__lt__(self,y)`
`x <= y`	`__le__(self,y)`
`x > y`	`__gt__(self,y)`
`x >= y`	`__ge__(self,y)`

Exercise – Rational Class

Write a class Rational that represents fractions, or Rational numbers.

The class should have the following data: `numerator` and `denominator`

The class should have the following methods:

- __init__
- Reduce(): reduces the fraction value
- Reciprocal(): finds the reciprocal of the value
- Special methods: add, subtract, multiply, divide, ==, <, >, str/repr

In the same file, but not inside the class, create a function that takes two numbers and returns the greatest common divisor using Euclid's algorithm.

Write a program that does the following:

- Creates two Rational numbers
- Does the following with the numbers: print, compare, add, subtract, multiply, divide and compare using less than and greater than.

Inheritance

Inheritance can be used to define new classes (child/derived/sub classes), by extending existing classes (parent/base/super classes).

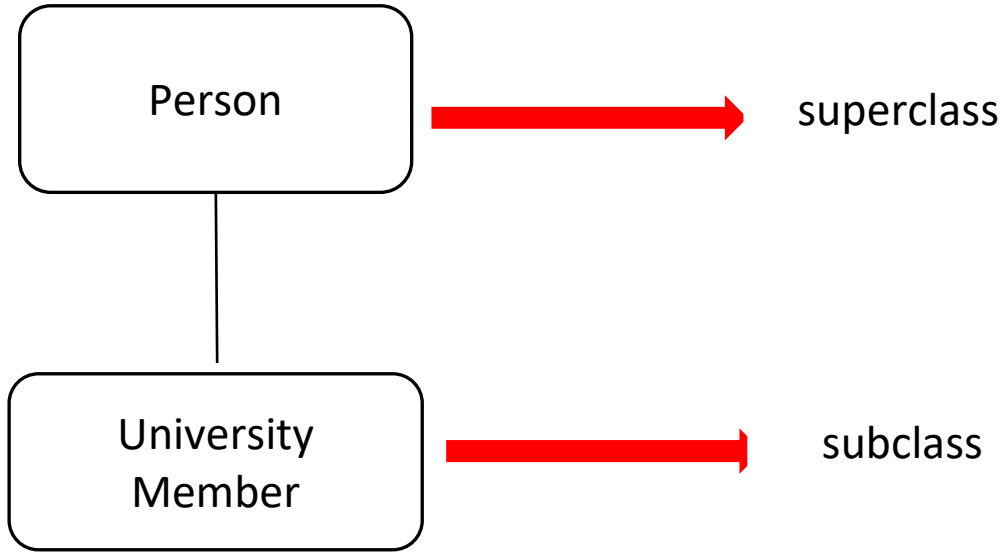
The subclass inherits properties and methods of the superclass in addition to adding some of its own properties and methods and overriding (modifying) some of the superclass methods.

The class `object` is at the top of the hierarchy, meaning every new class automatically inherits data and behaviors from the `object` class.

The subclass:

- Inherits attributes from its parent class, including all the methods / attributes in the `object` class.
- Can add new attributes.
- Can override or replace, methods / attributes of the superclass.

Example:



Defining Subclasses

Every new class is automatically a subclass of the object class, and it inherits default behaviors and functionality.

Create a subclass by specifying in parentheses after the class name the super class whose data and functionality it will inherit.

Example:

```
class myClass(superclass):
```

The subclass will extend the superclass by adding new or updated attributes and/or methods.

Method Overriding

The subclass inherits the methods from the superclass.

If you want to change the behavior of an inherited method, you can override it.

We can override inherited methods by specifying a new implementation in the subclass.

The methods in the subclass replace the functionality of the superclass method and implement different behavior.

When we invoke a method on an object, first the class of the object is examined for a matching method. If found the method is executed.

If there is no matching method in the subclass, the superclass(es) are searched until a match is found.

Class: UniversityMember

The `UniversityMember` class extends the `Person` class.

The class extends the superclass according to the following:
- Adds a new attribute, `idNum`.
- Overrides the `__init__`method.
- Overrides the `__lt__` method.

`str(p1)` in the code first checks to see if there is an `__str__` method in the class `UniversityMember`. Since there is not, it then checks to see if there is `__str__` method in its superclass `Person`. There is, so it uses that method.

`p1.getIdNum()` first checks to see if there is a `getIdNum` method in the class `UniversityMember`. There is, so it invokes that method.

07_UniversityMember.py

Class: UniversityMember (Version 2)

In this application we are comparing UniversityMembers.

Since `p1, p2,` and `p3` are all of type `UniversityMember`, the `__lt__` method defined in class `UniversityMember` will be invoked when evaluating the first two comparisons, so the ordering will be based on identification numbers.

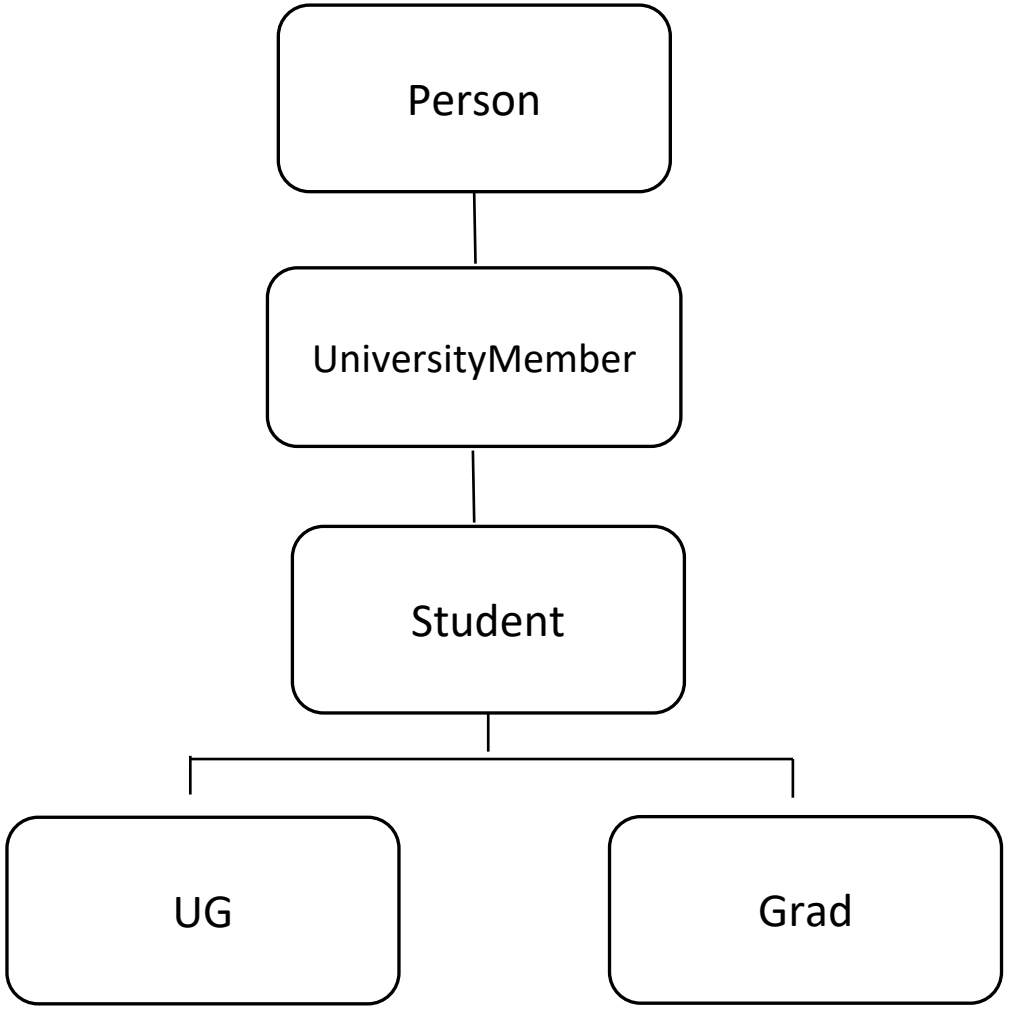
In the third comparison, the < operator is applied to operands of different types. `p4 < p1` is the shorthand for `p4.__lt__(p1).` So the `__lt__` method of `Person` class will be used, and the comparison with respect to the names will be performed.

What happens if we try

```
print ('p1 < p4 = ', p1 < p4)
```

See: 07_UniversityMember2.py

Multiple Levels of Inheritance



Reserved Word: `pass`

Indicates that the class the class has no attributes other than those inherited from its superclass.

Why do we create a class with no new attributes?

- See: 07_Student.py

`isinstance` built-in function

`isinstance(object, type)` returns `True` if and only if the first argument is an instance of the second argument.

eg. `isinstance([1,2], list)` is `True`

See: 07_Student2.py

Information Hiding and Inheritance

When a subclass tries to use a hidden attribute of its superclass, and AttributeError occurs.

In order to access inherited attributes, the subclass must call the get and set methods to access the hidden data.

Using information hiding can be difficult because of this, so often some Python programmers choose to not use the hidden attribute syntax (__)

Terms of Use

- This presentation was adapted from lecture materials provided in MIT Introduction to Computer Science and Programming in Python.
- Licenced under terms of Creative Commons License.