Objects and values

- In Python all values can be thought of as objects.
- Each object has a unique identity (an integer), a type (which is also an object) and a value that may be fixed or mutable.
- Objects can have attributes. These attributes can be read-only or writeable.
- Some value objects we have already studied are int, float, lists, tuples, strings, etc. The non-scalar values have various functional attributes (e.g. common functions on sequences).
- In Python classes allow a user to create user defined objects with specific behaviour.

Namespaces I

- A namespace is a set of name, value bindings. A namespace is looked up whenever a name is used and its value is needed (e.g. in expressions). To find a binding value the interpreter searches namespaces in the order defined by the scope rules - local, enclosing non-local, global.
- Namespaces are created at specific times during execution. For example the `__builtin__` and `__main__` module namespaces are created when the interpreter starts. As execution proceeds names get added to particular namespaces when specific statements are executed. For example: assignment, def statements, class statements, function invocation, class instantiation, import statements create/add name-value bindings to namespaces.
- The `del name` statement breaks the name-value binding from the local namespace. The name is removed from the local namespace.

Namespaces II

- Values are never destroyed. Reference (i.e. name-value) bindings may be removed. When a particular value has no references to the value it cannot be used any further and can be collected by the garbage collector.

Classes and objects I

- A class definition specifies the details of all its attributes (due to inheritance some/all of them could by in a different parent class). It is also a type.
- The attributes defined in a class can be data attributes or function attributes.
- A class is typically used to create instance objects that have the type of the class. These objects have access to the attributes defined in the class.
- The function attributes defined in the class are available as method objects.
- Just like function objects are created using the `def` keyword a class object is created using the `class` keyword. By convention the first letter of a class name is capitalized.

Classes and objects II

- When a class definition is executed it creates a *class object*. A class object allows references to attributes and instantiation (which creates object values of that type). The type of a class object is **type** since it is a type.
- It is important to distinguish between the class object and the instance objects created by the class object.

Inheritance I

- Inheritance is a mechanism by which already defined attributes in a class can be part of the definition of another class. The first class is often called the Baseclass and the second class is called the derived class. Since the base class can itself be a derived class we can in general have an inheritance hierarchy or tree.
- The derived class inherits the attributes of the base class (this happens recursively). Some or all of the attributes can be overidden.
- If no base class is specified then the special class **object** is assumed. So,all classes have **object** at the root of the inheritance hierarchy.
- Python allows multiple inheritance.
  `class Derived(Base1,Base2,...,BaseN):`
  The class Derived inherits from the base classes Base1 to BaseN.

Inheritance II

- When the value of an attribute is needed first the local class namespace is searched then the base class namespaces are searched in depth first, left to right order.

Object creation and access I

- An object is created by invoking the class name as a function. If C is a class:
  `o=C(...)`
  will create an object o of class C. This will invoke the __new__(cls[,...]) method most often inherited from **object** and create an empty object.
- The special method `__init__(self[,...])` called the initializer, if defined, is executed everytime an instance object is created. Note that this is not the constructor. A constructor called `__new__(cls[,...])` can be defined but most often the default one in **object** is used. The initializer is executed after the constructor has created the object to specify the initial object state - that is the initial values of its data attributes.

Object creation and access II

- Data attributes of an object are created only when they are assigned. So, base class data attributes are created only when the base class initializer is executed. This is usually done by `super().__init__(...)`.
- A class object creates a namespace for the attributes in the class when the class definition is executed. Similarly, an object created by instantiating a class also creates an instance specific namespace with the class name space as the enclosing namespace. All variables assigned using the statement `self.name=value` in the class definition are added to the name space of an instance when the assignment is executed.
- If a name is directly bound to a value inside the class (that is without using self) then it is a class attribute that is accessible to all instances of the class. All function attributes are accessible to all instance objects. When accessed they are converted to method objects and returned.

Object creation and access III

- If o is an object and we execute `o.a1=val1` then `a1` becomes an attribute of object o with value `val1` and will hide any attribute `a1` that is in the class object.

The class **object**

- The pre-defined class **object** is the root of the inheritance hierarchy. It defines many useful function objects that are inherited by all classes.
- For example, it defines: `__new__(cls[,...])`, `__str__()`, `__repr__()`, `__eq__()` amongst others. These are special functions and are called directly with the object as argument - for e.g. `str(o)`, `repr(o)` - without the normal '.' notation. The function `__eq__()` implements equality and is normally re-implemented for object equality for a particular class (otherwise it just checks object identity). Once implemented it will allow expressions like `o1==o2`, `o1!=o2`. Similarly, other comparators like `__le__()`, `__gt__()` etc. are also available and can be re-implemented if they make sense for the objects of a particular class.

Access and inheritance

- In an inheritance hierarchy it is possible that instance attributes and function attributes are overridden.
- To get access to overridden methods one can use two mechanisms:
  `<superclass>.function(o)` or `super(<class>, o).function()`, where `isinstance(o, <class>)` is True. `<superclass>, <class>` are class names.
- If an instance attribute and function name are simultaneously overridden and the function uses the instance attribute in the body then access will always be to the instance variable in the namespace of the object o and the instance variable in the superclass will not be accessible. (see code examples in the Python code).