Object Oriented Python and Decorators

Python Scopes and Namespaces

- A namespace is a mapping from names to objects
- Most namespaces are currently implemented as Python dictionaries
- Examples of namespaces are:
  - the set of built-in names - ValueError, max( ), etc.
  - global names in a module
  - local names in a function invocation
  - set of attributes of an object also form a namespace
- There is absolutely no relation between names in different namespaces
- references to names in modules are attribute references: in the expression - <modulename.functionname>
- __main__ : module of which statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of
- The local namespace for a function is created when the function is called, and deleted when the function returns
- A scope is a textual region of a Python program where a namespace is directly accessible
  - in case of nested scoping - go from inner most to the outermost

- When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace.
- the statements inside a class definition will usually be function definitions
- Function definitions bind the name of the new function here
- Class objects support two kinds of operations: attribute references and instantiation
- Attribute references use the standard syntax used for all attribute references in Python e.g. obj1.func1

- Class objects support two kinds of operations: attribute references and instantiation
- Attribute references use the standard syntax used for all attribute references in Python e.g. obj1.func1
- Class instantiation uses function notation; invokes the __init__ method
- The only operations understood by instance objects are attribute references
- There are two kinds of valid attribute names: data attributes and methods
- All attributes of a class that are function objects define corresponding methods of its instances
- A method is a function that “belongs to” an object.

- A method can be stored as an object (method object) for future use
- data attributes correspond to “instance variables”
- A data attribute could be assigned and deleted from an existing object
- nothing special about the name *self* but it is followed convention
- *self* is often first argument any function / method
- the methods / functions defined inside a class automatically pass *self* as the first argument
- Methods may call other methods by using method attributes of the *self* argument

Class and Instance Variables

- instance variables are for data unique to each instance
- class variables are for attributes and methods shared by all instances of the class

Iterators

- How a for loop works on an iterable object such as a list:
  - Behind the scenes, the for statement calls iter() on the container object
  - iter() returns an iterator object
  - iterator object defines the method __next__() which accesses elements in the container one at a time
  - When there are no more elements, __next__() raises a StopIteration exception which tells the for loop to terminate

Making a class Iterable

- implement __iter__ and __next__ methods

Making a class printable

- implement the __str__ method

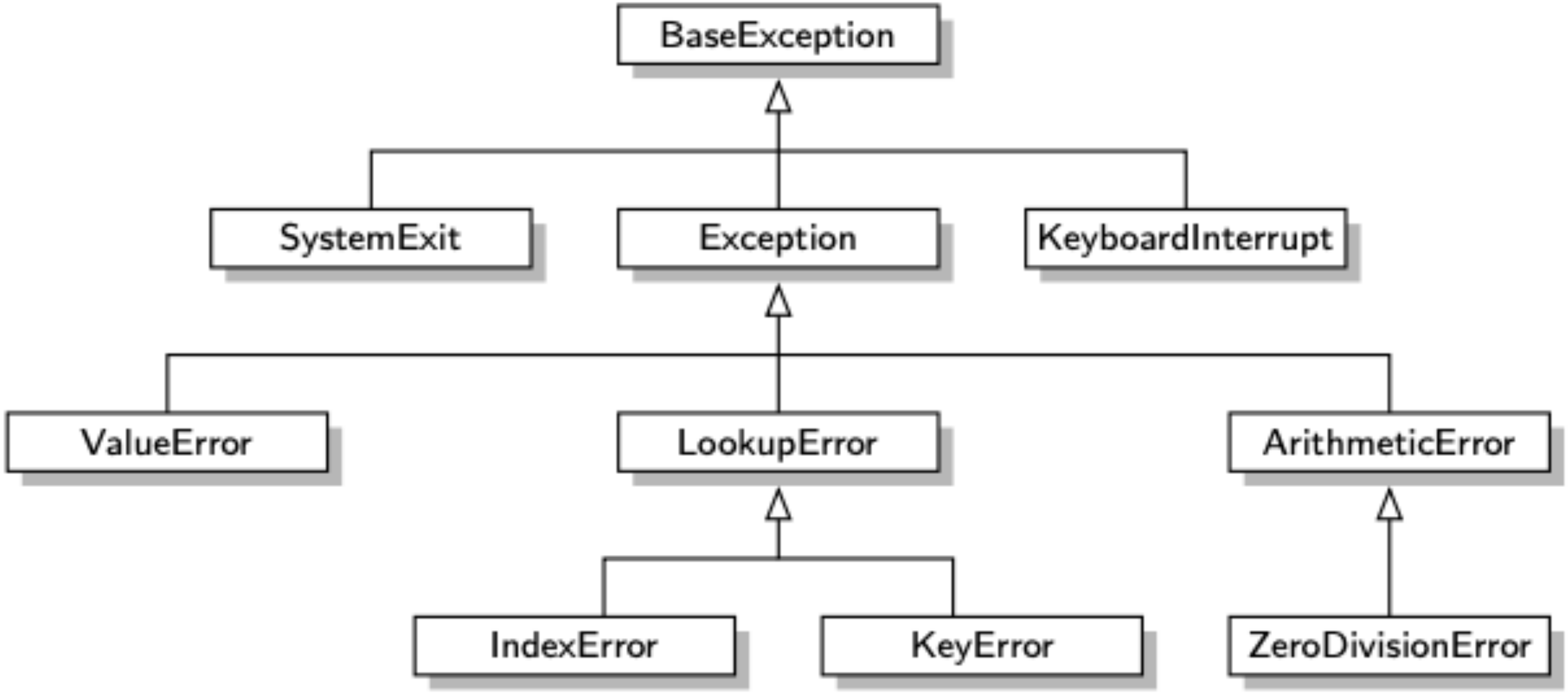
Inheritance

- if a requested attribute is not found in the class, the search proceeds to look in the base class
- Derived classes may override methods of their base classes
- An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name
- There is a simple way to call the base class method directly: just call BaseClassName.methodname(self, arguments)
- Python has two built-in functions that work with inheritance
    - isinstance()
    - issublcass()

Inheritance

- A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method
- A subclass may also extend its superclass by providing brand new methods
-

Inheritance - Example



Inheritance

- The body of the derived class might have new constructor which relies upon making a call to the inherited constructor to perform most of the initialization - reply on super( )
- Other overriding methods could make a call to the base class's method again using super( )

Making a class callable

- Used to make an instance of a class behave like a function, i.e. the instance object behaves like a function.
  - useful for contexts where a function is expected and not just a "non-function" object
  - useful as function decorators
- Use the __call__ method implementation makes a class callable

Callable

- All functions are callable
- use callable( ) to know if an object is callable
- classes are callable e.g. str, int, etc. and your own custom classes
- instances of class are NOT callable
- to make instance callable one would need to implement the dunder (short for double underscore) method __call__

Callable class and instance

- From a python shell do the following:
  - import savingsbankaccount_v4
  - callable(savingsbankaccount_v4.SavingsBankAccount)
  - from savingsbankaccount_v4 import SavingsBankAccount
  - sbs = SavingsBankAccount(1000,’pradip’,200)
  - callable(sbs)
- Now try the above using the class and objects from version 5

Function Decorators

- A function decorator takes another function as an argument and returns a decorated function
- It enables modification / extension of an existing function without any changes to the original function source code
- There are also class decorators (out of current scope)

Decorator implementation

- use of @ symbol to decorate
- Function decorators
  - Decorating a function with nested function decorator
  - Decorating a function with a callable class decorator - use of __call__ method implementation
  - See examples - nested_function_decorator.py, mycallableclass_v1.py, mycallableclass_v2.py