The UK's European university

University of Kent

COMP8270 / PROGRAMMING FOR ARTIFICIAL INTELLIGENCE

Fernando Otero
febo@kent.ac.uk
cs.kent.ac.uk/people/staff/febo

overview:

1. Inheritance

Scope and Namespaces:

- Class definitions define namespaces, which are related to Python's scope rules
  - We need to know them in order to make sure we are using the correct variables/functions
- namespace = mapping from names to objects
- Let's consider an example…

Scope and Namespaces:

```
def scope_test():
    def do_local():
        spam = "local spam”      # local (inner function) scope

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam”  # nonlocal (outer function) scope

    def do_global():
        global spam
        spam = "global spam”    # global scope

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Scope and Namespaces:

```
def scope_test():
    def do_local():
        spam = "local spam”     # local (inner function) scope

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam”  # nonlocal (outer function) scope

    def do_global():
        global spam
        spam = "global spam”    # global scope

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
```

```
In global scope: global spam
```

Scope and Namespaces:

```
class Dog:

    kind = 'canine'          # class variable shared by all instances

    def __init__(self, name):

        self.name = name     # instance variable unique to each instance


d = Dog('Fido')
e = Dog('Buddy')

print(d.kind)                # shared by all dogs
# 'canine'

print(e.kind)                # shared by all dogs
# 'canine'

print(d.name)                # unique to d
# 'Fido'

print(e.name)                # unique to e
# 'Buddy'
```

Scope and Namespaces:

- If the same attribute name occurs in both an instance and in a class, then attribute lookup prioritizes the instance

```
class Dog:
    kind = 'canine'         # class variable shared by all instances
    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

    def update_kind(self, k):
        self.kind = k       # instance variable unique to each instance


d = Dog('Fido')
print(d.kind)               # shared by all dogs
# 'canine'
d.update_kind('poodle')
print(d.kind)               # unique to d
# 'poodle'
print(Dog.kind)             # shared by all dogs
# 'canine'
```

Class Inheritance:

- The syntax for a derived class definition looks like this:

superclass

```
class DerivedClassName(BaseClassName):
    # <statement-1>
    # .
    # .
    # .
    # <statement-N>
```

- Base class is used for resolving attribute/method references:
  - if a requested attribute/method is not found in the class, the search proceeds to look in the base class
  - this rule is applied recursively if the base class itself is derived from some other class.

Example (I):

```
class Super:
    def method(self):
        print("in Super.method")

class Sub(Super):
    def method(self):
        print("starting Sub.method")
        Super.method(self)
        print("ending Sub.method")

x = Super()
x.method()
x = Sub()
x.method()
```

```
in Super.method
```

```
starting Sub.method
in Super.method
ending Sub.method
```

Example (2):

```
class Super:
    def method(self):
        print("in Super.method")

class Sub(Super):
    def method(self, value):
        print("starting Sub.method")
        Super.method(self)
        print("ending Sub.method")

x = Super()
x.method()

x = Sub()
x.method()
```

```
in Super.method
```

```
?
```

Notes:

- You must call the `Super.__init__`[1] method explicitly if you want it to run

- No attribute/method qualifiers private, public, protected
  - "Private" method/attributes are prefixed with one or two `"_"` characters

- How do we define properties?

---

[1] **Notation**: `super().__init__(`*parameter values*`)`

Properties:

- Decorator @property
  - Turns a method into a “getter” for a read-only attribute with the same name

```
class Socket:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        print("in Socket.voltage")
        return self._voltage
```

- We can also specify **setter** and **deleter** methods

Properties:

```
class Socket:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        print("getter Socket.voltage")
        return self._voltage

    @voltage.setter
    def voltage(self, value):
        print("settter Socket.voltage")
        self._voltage = value

    @voltage.deleter
    def voltage(self):
        print("deleter Socket.voltage")
        del self._voltage
```

Properties:

- You could still do this:
    - …but that would be morally wrong!

```
s = Socket()
s._voltage = 0
```

Multiple Inheritance:

- Python support multiple inheritance!

```
class DerivedClassName(Base1, Base2, Base3):
    # <statement-1>
    # .
    # .
    # .
    # <statement-N>
```

- Clashes in names:
  - attributes/method are inherited from a parent class as depth-first, left-to-right

Multiple Inheritance:

```
class A:
    def method(self):
        print("in Super.A")

class B:
    def method(self):
        print("in Super.B")

class C(B, A):
    pass              # no extra implementation

x = C()
x.method()
```

- What value gets printed?

Abstract Classes:

- Abstract classes are created using the module `abc`:

```
from abc import ABC, abstractmethod

class MyAbstractClass(ABC):
    @abstractmethod
    def method(self):
        # needs to be implemented
        pass
```

- You can't instantiate an abstract class if it contains abstract methods
    - only subclasses of `ABC` can have abstract methods

Useful functions (1):

- `isinstance()`: checks if an instance's type is the same as another class or it is a derived class from it

```
obj = 5
isinstance(obj, int)    # True
obj = 5.5
isinstance(obj, float) # True
```

- `issubclass()`: checks if a type is a subclass of another type

```
issubclass(float, int)     # False
issubclass(float, object) # True
```

Useful functions (2):

- `type()`: returns the type of an object
    - in most cases the same as `object.__class__`

```
a = 5
type(a)      # int
alist = ["1", "2", "3"]
type(alist) # list
```

Next lecture:

- Exceptions
- Arrays with NumPy