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. Classes
2. Attributes
3. Methods
4. Constructors

Python Classes

A Common Programming Problem

```
BankAccount1 = [100]
Name1 = "Sally”

BankAccount2 = [1500]
Name2 = "Tariq"

def Increment(Account, amount):
    Account[0] += amount

def Decrement(Account, amount):
    Account[0] -= amount

def PrintAccount(*AccountInfo):
    for u in AccountInfo:
        print (u)

Decrement(Name1, 200)                          # Run-time error
PrintAccount(BankAccount1, Name2)              # Semantically incorrect
```

- What we want is a type that encapsulates a bank account.
    - Manipulates internal state correctly.
    - Protects internal state from unwanted mistakes.
- Programmer defined types give us that.

Object Oriented Programming

- The problem is we need to associate a number of variables as they are meaningful together.
  - The owner and the balance are ”meaningless” on their own.
- Primitive types do not give us that.
- The idea is to support programmer defined types and their operations.
  - Collect the variables we need in one place and treat them as one.
- Python uses classes to support user defined types.
- Python classes encapsulate data and the functions needed to safely manipulate them.

The Python Class

```
class class-name:
<tab>   member
<tab>   member

member = method | attribute
```

- Python supports programmer defined types with `class`.
- A `class` groups together (encapsulates) data and functions.
- Once a class has been defined it can be instantiated; the instantiation of a class is called an object.
- An object is a concrete instance of a class.
    - 1 class, there can be many objects

Attributes

Class Attributes (Our Type’s Data)

```
class Account:
       Balance = 0
       Owner = None


MyAcct = Account()                # Create an object of type Account
MyAcct.Balance = 1000000          # Assign the account balance
MyAcct.Owner = “Doug”             # Assign the owner
```

- Class attributes are the data that the type needs.
- You, the programmer, decide what you need to solve your problem.

More on Attributes

```
MyAcct = Account()                  # Create an object of type Account
MyAcct.__sizeof__() → 32            # Size in bytes of our class

dir(MyAcct) → ['Balance', 'Owner', '__class__', '__delattr__',
'__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__’]
```

- The Python built-in, `dir()`, shows us all the attributes of an object.
- The `dir()` function works on all objects.
- It returns a list that we can use like any other.
- Attributes are how Python polymorphism works.

Methods

Class Methods

- Our type has data, but we need to operate on it.
- We use class methods to operate on class data.
- Class methods are defined like normal functions inside the scope of a class.
- Unlike other languages, the instance reference is an explicit argument to the method.
  - In Java and C++ “this” is implicit.
  - In Python the current object is called “self”, and it must appear in the function header.

More on our Account Class

```
class Account:
        Balance = 0
        Owner = None

        def UpdateBalance(self, amount):
                VerifyOK = self.Balance + amount        # Local var
                Balance = VerifyOK                      # Local var
                self.Balance = VerifyOK                 # our object

MyAcct = Account()
MyAcct.UpdateBalance(1000)                              # Credit 1,000
MyAcct.UpdateBalance(-200)                              # Debit 200
```

- Note the `self` is declared in the function header.
- `self` is not specified in the invocation.

A Safer Way

```
class Account:
          Balance = 0
          Owner = None

          def IncrBalance(self, amount):
                    self.Balance += amount

          def DecrBalance(self, amount):
                    self.Balance -= amount

MyAcct = Account()
MyAcct.IncrBalance(1000000)
```

- This is a safer version to update the balance of an account.
- A method argument should not determine the behaviour of a method.
  - Defensive programming
- Three specific functions is safer than one function with three different behaviours.
  - More semantic checks that can be relied upon.

Another Example: A Stack

```
class Stack:
    Stack = list()

    def push(self, x):
        self.Stack.append (x)

    def pop(self):
        return self.Stack.pop ()

Z = Stack()
Z.push(1)
Z.push(2)
Z.push(x = 3)
for i in range(1, 4):
    print (Z.pop())
3
2
1
```

The Class Object

```
def Show(self):
       print(self.Owner, self.Balance)


Acct0 = Account(“Doug”, 1000)
Account.Show = Show                     # Modify “class” object
Account.Display = Show                  # The name does not matter
Acct1 = Account(“Lene”, 5000)
Acct0.Show() → Doug 1000                # Existing objects get it
Acct0.Display() → Doug 1000
Acct1.Show() → Lene 5000
```

- We can also add a method after the class definition.
- New and existing objects receive the update.
- The class definition is itself an object of type “class”

The Class Object Con’t

```
def factory(a_type):
        return a_type()


x = factory(int)
x → 0                         # Initial value
type(x) → <class 'int’>       # Object type
TypeList = [int, str, Account]
→[<class 'int'>, <class 'str'>, <class '__main__.Account’>]
```

- A class object is like any object. We can:
    - Pass it as an argument.
    - Include it in a list.
- Almost everything in Python is an object with a type.

Constructors

Class Constructors

- When we create instances of classes, we need to initialize them.
- The caller should not have to do it; the class should know how to initialize itself.
- Python provides for that with a special class method: `__init__()`
- Python calls the `__init__()` function of a class after it is created, and before the object is returned to the caller.
    - So the memory is there and ready to go.

The Account Constructor

```
class Account:
            def __init__(self, Owner, Balance):
                        self.Owner = Owner
                        self.Balance = Balance

            def IncrBalance(self, amount):
                        self.Balance += amount

            def DecrBalance(self, amount):
                        self.Balance -= amount

MyAcct = Account(“Doug”, 1000)
MyAcct.IncrBalance(100)
print(MyAcct.Balance)
→ 1100
AnotherAcct = Account(Owner = “Tariq”, Balance = 1000000)
Broken = Account()                                          # Error, missing arguments
```

- Notice that we have created the Owner and Balance attributes differently.
    - This is important.

The Multiple Constructors

```
class Account:
        def __init__(self):
                self.Owner = None
                self.Balance = -1
                return
        def __init__(self, Owner, Balance):
                self.Owner = Owner
                self.Balance = Balance


Acct = Account(“Sally”, 1000)
Broken = Account()                      # Still Broken
```

- There can only be one __init__.
- Python will accept the above class definition, but the last __init__ it sees is the one it will use.
    - We are simply over-writing the name, __init__

The Multiple Constructors: Solution

```
class Account:
        def __init__(self, *args):
                N = len(args)
                if N == 0:
                        self.Owner = None
                        self.Balance = -1
                        return
                elif N == 2:
                        self.Owner = Owner
                        self.Balance = Balance
                        return
                else
                        # Error, we will learn how to raise an error in later lectures

Acct = Account(“Sally”, 1000)
Works = Account()                            # Works!
Broken (“Frank”)                             # Broken
```

- Create a single __init__ with multiple behaviours.
    - Yes, this is what I suggested you avoid a few slides ago.

Class Versus Instance Attributes

```
class Account:
         Overdraft = -1000          # Per Class

         def __init__(self, Owner, Balance):
                 self.Owner = Owner          # Per instance
                 self.Balance = Balance      # Per instance

Account.Overdraft → -1000
MyAcct = Account(“Doug”, 1000)
MyAcct.Balance → 1000
Account.Overdraft = -2000          # All Account objects see this change
MyAcct.Overdraft = -3000           # Over-ride class in instance
```

- Attributes created in the constructor are called *instance* attributes.
- Attributes declared at the class level are *class* attributes.

Methods Revisited

```
class Account:
        OverDraft = -1000

        def __init__(self, Owner, Balance):
                self.Owner = Owner
                self.Balance = Balance

        # Method on an object, has access to object’s self
        def InstanceMethod(self):
                print (self.Owner, self.Balance)

        # Method on the class, has access to the class object
        @classmethod
        def ClassMethod(cls):
                print(cls.OverDraft)

        # Method with access to nothing, an association
        @staticmethod
        def StaticMethod():
                print (“I have access to nothing.”)
```

An Example

```
class RecommenderSystem:
    data = None         # Every instance of the class can use this

    def __init__(self, HyperParameter):
        self.Parameter = HyperParameter

    def BuildModel(self):
        # We have created a new object attribute in a method
        self.Model = BuildCluster (self.data, self.Parameter)

    @classmethod
    def SetupRun(cls, filename):
         RecommenderSystem.data = LoadData(filename)

    @staticmethod
    def LoadData(filename):
        return DoSomeWork ()

RecommenderSystem.SetupRun(“Data-10-11-2021”)
Experiment0 = RecommenderSystem(0.1)
Experiment1 = RecommenderSystem(0.5)
```

Next lecture:

- Inheritance