Object Oriented Programming in Python

- PoPI

Objects

- Object-oriented programming is a particular methodology of programming (approach to organise code)
- Widely used since 90s
- Code is written around an object, which is
  - data – a current state
  - methods – operations that can access/change the state
- Example: cash register id 2346
  - data:
    - number of items currently registered/counted (7)
    - total price (£32)
  - methods:
    - add an item with a given price
    - check the total price ("Your total is N")
    - check the number of items ("You have M items")
    - clear the register (before starting to process next operation)

Objects (cont)

def main():

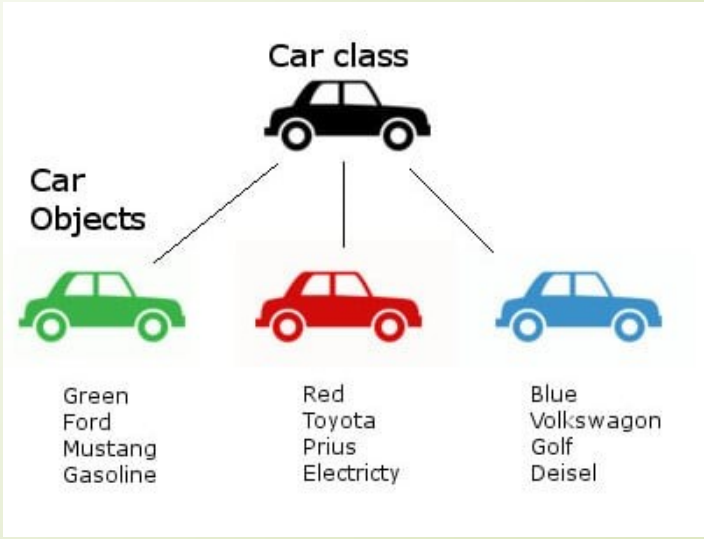
- register2346.addItem(15.56) #adds an item of that price
- register2346.addItem(12.10) #adds another item
- M = register2346.getCount() # M gets value 2 as we have 2 items
- N = register2346.getTotal() # N gets value 17.66 as it is the total
- register2346.clear() # clear the data (set num of items and total to 0)

- regster2346.addItem(5.05)
- register5375.addItem(3.34) # here we work with another register of the same type

What does it mean that regster2346 and register5375 are of the same type?

Classes

- A class is a recipe or a "blueprint" for creating (many) objects
  - such as register2346, register5375, register10000,…
  - they are called instances of a class
  - a class can be named CashRegister
- A class is specified by
  - variables names to store state/data of each object of the class
  - methods specified by
    - method names
    - required arguments to call them
    - their implementations



Class Example

```
class CashRegister :
    def clear(self) :                    #  method
        self._itemCount = 0              #  variable to store number of items
        self._totalPrice = 0.0           #  variable to store total

    def addItem(self, price) :           #  method
        self._itemCount = self._itemCount + 1
        self._totalPrice = self._totalPrice + price

    def getTotal(self) :                 # method
        return self._totalPrice

    def getCount(self) :                 # method
        return self._itemCount
```

Class Example (cont.)

- Creating objects as instances of a class:

```
def main():
    register2346 = CashRegister()      #an instance of the class (=object) created
    register5375 = CashRegister()      #another instance of the class (=object) created
    register2346.addItem(5.05)
    register5375.addItem(3.34)
    …
```

Constructors: Motivation

- The previous program won't run (as well as this one)

  register2346 = CashRegister()

  register2346.addItem(5.05)

- Why?
- It would run if modified

  register2346 = CashRegister()

  register2346.clear()

  register2346.addItem(5.05)

- The variables _itemCount and _totalPrice were not initialised (by 0 and 0.0)
- To automatically initialise such variables when an object is created

  register2346 = CashRegister()

  use constructors

Constructors

- Has a reserved name __init__ but arguments may vary

```
class CashRegister :
    def __init__(self) :
        self._itemCount = 0
        self._totalPrice = 0.0
    def …
```

- Alternatively, to the first __init__ we can define

```
    def __init__(self, discount) :
        self._itemCount = 0
        self._totalPrice = - discount
```

- We can then do in the main program depending on what we defined:

  register2346 = CashRegister() or register2346 = CashRegister(50)

- Constructors can perform any actions inside the body but they must, first, initialise all the variables that describe state/data of objects of this class
- Though __init__ defined as a method never call it explicitly

  register2346.__init__()

Methods and Self

- You define a method of a class

  def methodname(self, arg1, ..., argN)

- And you call it in the main program as follows

  object.methodname(arg1,..., argN)

- What if we want to call a method of a class from another method

  class CashRegister :

  def addItem(self, price):

  call getTotal() to check current total and only add if it is <=100

  - to which object should I "talk" to to getTotal()?
  - to this object itself, therefore we write

  class CashRegister :

  def addItem(self, price):

  tot = self.getTotal()

  if tot <=100: self._itemCount = self._itemCount + 1 …

Methods and Self (cont)

- It is the same with variables

```
class CashRegister :
    def clear(self) :
        self._itemCount = 0          # update _itemCount of this object itself
```

- Sometimes we need to distinguish object itself from other:

```
class CashRegister :
    def copy(self, other) :          #copy the state of other cash register to this one
        self._itemCount = other._itemCount
        self._totalPrice = other._totalPrice
```

- In the main program

```
...
register2346.copy(register5375)
```

Encapsulation: Motivation

- A physical cash register manufacturer wants you to operate the register in an expected way (pressing buttons, switching on/off)
- He doesn't want you to open the register and change its components because this may lead to inconsistent behaviour
- In the same way, a developer of CashRegister class wants you to use only in safe way by either: clear it, adding an item with a price, and receiving info from getTotal() and getCount()
- CashRegister class developer doesn't want you to touch the variables _itemCount and _totalPrice in the main program because you can cause inconsistency

```
def main():
    ...
    register2346._totalPrice = 1000      # setting a high price to pay without
                                         adding any purchased item info
```

Encapsulation

- In Java, C++ and other languages we can declare that some variables/methods are restricted (cannot be accessed from functions/methods outside of the defining class)

```
class CashRegister
{
    private int _totalPrice
}
```

- In Python no restrictions but a convention:
  - Methods and variables of the shape _name are restricted/private
  - You are not supposed to call them outside of the class definition code

```
def main():
    …
    register2346._totalPrice = 1000
```

Assigning Object Variables

- Consider the following code

```
register2346 = CashRegister()
register5375 = CashRegister()
register2346.addItem(5.05)
register5375.addItem(5.05)          # thus, objects register2346 and
                                      register5375 have identical content

print(register2346 == register5375)
>> False
```

- Why?
- When register2346 = CashRegister() is executed:
  - A new piece of memory is allocated
  - A reference to that memory is stored in the variable register2346 (recall lecture on Memory and References)
  - register2346 and register5375 point to different pieces of memory and therefore are not equal

Assigning Object Variables (cont.)

- To check for content equality:

```
class CashRegister :
    def isEqual(self, other):
        if self._itemCount == other._itemCount and self._totalPrice == other._totalPrice:
            return True
        else: return False
    …
def main():
    register2346 = CashRegister()
    register5375 = CashRegister()
    register2346.addItem(5.05)
    register5375.addItem(5.05)
    print(register2346.isEqual(register5375))
>> True
```

Operators Overloading

- If you want == to work in the same way as isEqual(self, other) you can!
  - Use operator overloading feature
- isEqual should change to have a standard name __eq__, two arguments self and y, and return True or False i.e.,

```
class CashRegister :
    def __eq__(self, y):
        if self._itemCount == y._itemCount and self._totalPrice == y._totalPrice:
            return True
        else: return False
```

- Then we can do:

```
register2346 = CashRegister()
register5375 = CashRegister()
print(register2346 == register5375)
>> True
```

Operators Overloading (cont.)

- You can overload not only ==

Expression	Method Name	Returns	Description
x + y	__add__(self, y)	object	Addition
x - y	__sub__(self, y)	object	Subtraction
x * y	__mul__(self, y)	object	Multiplication
x / y	__truediv__(self, y)	object	Real division
x // y	__floordiv__(self, y)	object	Floor division
x % y	__mod__(self, y)	object	Modulus
x ** y	__pow__(self, y)	object	Exponentiation
x == y	__eq__(self, y)	Boolean	Equal
x != y	__ne__(self, y)	Boolean	Not equal
x < y	__lt__(self, y)	Boolean	Less than
x <= y	__le__(self, y)	Boolean	Less than or equal
x > y	__gt__(self, y)	Boolean	Greater than
x >= y	__ge__(self, y)	Boolean	Greater than or equal
-x	__neg__(self)	object	Unary minus
abs(x)	__abs__(self)	object	Absolute value
float(x)	__float__(self)	float	Convert to a floating-point value
int(x)	__int__(self)	integer	Convert to an integer value
str(x) print(x)	__repr__(self)	string	Convert to a readable string
x = *ClassName*()	__init__(self)	object	Constructor

In Python Every Data Type is a Class

- Lists

```
lst = list([1, 2, 3])          # constructor
lst1 = [4,5,6]                 # the same as previous, simplified syntax
lst.append(lst1)               # a method
```

  - Operators [] are also overridden, e.g., lst[2] could be lst.__getItem__(2)

- Strings

```
str = string("abc")            # constructor
str1 = "def"                   # the same as previous, simplified syntax
str2 = str1 + str2         # effectively str2 = str.__add__(str2) overridden
```

- Integers

```
a = 3                          # constructor
b = a + 2                      # effectively b = a.__add__(2) overridden
```

- Floats

```
f = 1.0                        # constructor
print(f.is_integer())
>> True
```