Custom objects
Emulating numeric types

Lecture 07.02

By Marina Barsky

Special type: Time

Starter code

Class *Time*

- Attributes:
  - hours and minutes

- Methods:
  - Add time
  - Subtract time
  - Compare time (for sorting)

time_solution.py

Modeling Cash Registers

cash_register.py
cash_register_special.py

Cash register class – blueprint for creating new cash registers

```
if __name__ == '__main__’:
    # A cash register with 5 ones, 5 twos,
    # 5 fives, 5 tens, and 5 twenties,
    # for a total of $190.
    register = CashRegister(5, 5, 5, 5, 5)
    print(register.get_total())

    register.add(3, 'twos')
    register.remove(2, 'twenties')

    print(register.get_total())
```

Defining Class *CashRegister*

- The first line of the class definition is:

```
class CashRegister:
```

Constructor

Constructor, called to initialize an object. By convention, the first parameter is ***self***. It refers to the *CashRegister* object that is being initialized

```
class CashRegister:
  def __init__(self, ones, twos, fives, tens, twenties):
        self.ones = ones
        self.twos = twos
        self.fives = fives
        self.tens = tens
        self.twenties = twenties
```

creates an ***instance variable*** *ones* that belongs to the *CashRegister* object

Variables belonging to an object are often called its ***fields*** or ***attributes***

We can already use our new type to create cash registers

```
if __name__ == '__main__’:
    register1 = CashRegister(5, 5, 5, 5, 5)
    print (register1.tens)
    register1.twenties = 6

    register2 = CashRegister(5, 5, 5, 5, 6)


    print (register1 is register2)
    print (register1 == register2)
```

Two **different** objects of the same class

```
False
False
```

Adding capabilities: method *add()*

```
def add(self, count, denomination):
    """ (CashRegister, int, str) -> NoneType

    Add count items of denomination to the register.
    denomination is one of 'ones', 'twos',
    'fives', 'tens', and 'twenties’.
    """

    if denomination == 'ones':
        self.ones += count
    elif denomination == 'twos':
        self.twos += count
    elif denomination == 'fives':
        self.fives += count
    elif denomination == 'tens':
        self.tens += count
    elif denomination == 'twenties':
        self.twenties += count
```

Adding capabilities: method *get_total*

self refers to a particular register whose total is being asked for

```
def get_total(self):
    """ (CashRegister) -> int

    Return the total amount of cash in the register.

    >>> register = CashRegister(5, 5, 5, 5, 5)
    >>> register.get_total()
    190
    """

    return self.ones + self.twos * 2 + self.fives * 5 + \
             self.tens * 10 + self.twenties * 20
```

Exercise

- Based on a code provided in file *cash_register.py*, implement method *remove* according to its docstring.
- Start from copying an existing method *add*, and make a couple of changes
- Run the program and see if your *remove* method works as expected

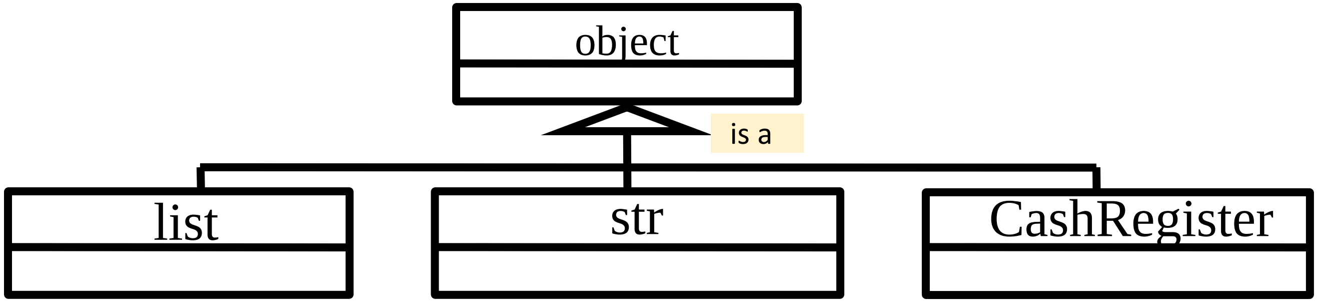
- To think about:

How would you modify internal representation of cash denominations in order to make your code more expressive and concise?

Making our own classes ‘members of the Pythonic society’

- When we add a new data type defined in our own class, we want it to behave in the same way as other Python types:
  - Print CashRegister object
  - Add 2 cash registers using +
  - Compare 2 cash registers for equality using ==
  - …

Everything is an object



- All different types of objects inherit methods from a very basic root class *object*
- These basic methods are implemented in the *object* class

```
dir(object)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

‘Useless’ printing

```
cr1 = CashRegister(2, 0, 0, 0, 0)
cr2 = CashRegister(0, 1, 0, 0, 0)
cr3 = CashRegister(1, 1, 0, 0, 0)
```

```
print(cr1)
print(cr3)
```

```
<__main__.CashRegister object at 0x000001D09FF36390>
<__main__.CashRegister object at 0x000001D09FF36400>
```

- Special method **`__str__`** is called to get a string representation of an object (**`str()`** or **`print()`**)

- But our **`CashRegister`** does not have code for **`__str__`** - so the **`__str__`** method of an ***`object`*** class is used instead

We need our own __str__

```
def __str__(self):
    """ (CashRegister) -> str
    Return a string representation of this CashRegister.
    >>> reg1 = CashRegister(1, 2, 3, 4, 5)
    >>> reg1.__str__()
    CashRegister: $160 ($1x1, $2x2, $5x3, $10x4, $20x5)
    """
```

We want to see **that** when the __str__ method is called

Implementing our own __str__ (1/3)

```
def __str__(self):
    """ (CashRegister) -> str
    Return a string representation of this CashRegister.
    >>> reg1 = CashRegister(1, 2, 3, 4, 5)
    >>> reg1.__str__()
    CashRegister: $160 ($1x1, $2x2, $5x3, $10x4, $20x5)
    """

    return 'CashRegister: $’ + \
      self.get_total() + ' ($1x' + self.ones + \
      ', $2x' + self.twos + ', $5x' + self.fives + \
      ', $10x' + self.tens + ', $20x’ + \
      self.twenties + ')'
```

Will this work?

Implementing our own __str__ (2/3)

```
def __str__(self):
    """ (CashRegister) -> str
    Return a string representation of this CashRegister.
    >>> reg1 = CashRegister(1, 2, 3, 4, 5)
    >>> reg1.__str__()
    CashRegister: $160 ($1x1, $2x2, $5x3, $10x4, $20x5)
    """

    return 'CashRegister: $’ + \
      str(self.get_total()) + ' ($1x’ + str(self.ones) + \
      ', $2x’ + str(self.twos) + ', $5x' + str(self.fives)+\
      ', $10x' + str(self.tens) + ', $20x’ + \
      str(self.twenties) + ')'
```

This code is **extremely ugly** and error-prone! What to do?

Implementing our own __str__ (3/3) – using **format**

```
def __str__(self):
    """ (CashRegister) -> str
    Return a string representation of this CashRegister.
    >>> reg1 = CashRegister(1, 2, 3, 4, 5)
    >>> reg1.__str__()
    CashRegister: $160 ($1x1, $2x2, $5x3, $10x4, $20x5)
    """

 return 'CashRegister: ' + \
  '${0} ($1x{1}, $2x{2}, $5x{3}, $10x{4}, $20x{5})'.format(
      self.get_total(), self.ones, self.twos,
      self.fives, self.tens, self.twenties)
```

Placeholders for actual values

Actual values to substitute

Now we can print cash registers

```
cr1 = CashRegister(2, 0, 0, 0, 0)
cr2 = CashRegister(0, 1, 0, 0, 0)
cr3 = CashRegister(1, 1, 0, 0, 0)

print(cr1)
print(cr3)
```

```
CashRegister: $2 ($1x2, $2x0, $5x0, $10x0, $20x0)
CashRegister: $3 ($1x1, $2x1, $5x0, $10x0, $20x0)
```

Optional method: __repr__

__str__ ("dunder* - string") and **__repr__** ("dunder-repper") are both special methods that return strings representing the state of the object

__repr__ provides backup behavior if **__str__** is missing (that is - it is enough to implement __repr__)

__repr__ is a printable representation of an object for programming and debugging

__str__ is a nicely printable representation of an object for the user of your program

*double-underscore

Implementing __repr__ is important to print *list of objects*

```
cr1 = CashRegister(2, 0, 0, 0, 0)
cr2 = CashRegister(0, 1, 0, 0, 0)
cr3 = CashRegister(1, 1, 0, 0, 0)

crs = []
crs.append(cr1)
crs.append(cr2)
crs.append(cr3)

print(crs)
```

Without __repr__:

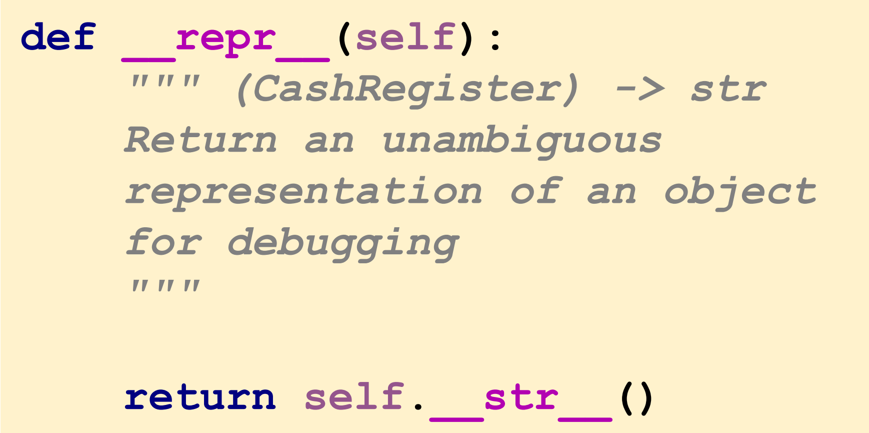
```
[<__main__.CashRegister object at 0x000001E8A63966A0>,
<__main__.CashRegister object at 0x000001E8A63964A8>,
<__main__.CashRegister object at 0x000001E8A63964E0>]
```

With __repr__ implemented

```
cr1 = CashRegister(2, 0, 0, 0, 0)
cr2 = CashRegister(0, 1, 0, 0, 0)
cr3 = CashRegister(1, 1, 0, 0, 0)

crs = []
crs.append(cr1)
crs.append(cr2)
crs.append(cr3)

print(crs)
```



```
def __repr__(self):
    """ (CashRegister) -> str
    Return an unambiguous
    representation of an object
    for debugging
    """

    return self.__str__()
```

```
[CashRegister: $2 ($1x2, $2x0, $5x0, $10x0, $20x0),
CashRegister: $2 ($1x0, $2x1, $5x0, $10x0, $20x0),
CashRegister: $3 ($1x1, $2x1, $5x0, $10x0, $20x0)]
```

Comparing two cash registers using ==

```
help (object.__eq__)
Help on wrapper_descriptor:
__eq__(self, value, /)
    Return self==value.
```

- We implement the `__eq__` method to our *CashRegister* class so that we can compare two cash register objects using `==`
- **Our decision:** We will consider two cash registers to be equal if they contain the same total amount of cash

Implementing __eq__

```
def __eq__(self, other):
    """ (CashRegister, CashRegister) -> bool
    Return True iff this CashRegister
    has the same amount of money as other.
    >>> reg1 = CashRegister(2, 0, 0, 0, 0)
    >>> reg2 = CashRegister(0, 1, 0, 0, 0)
    >>> reg1 == reg2
    True
    """

    return self.get_total() == other.get_total()
```

Now we can compare

```
cr1 = CashRegister(2, 0, 0, 0, 0)
cr2 = CashRegister(0, 1, 0, 0, 0)
cr3 = CashRegister(1, 1, 0, 0, 0)
```

In class CashRegister:

```
    def __eq__(self, other):
        return self.get_total() == other.get_total()
```

self becomes left operand (**cr1**),
other becomes right operand (**cr2**)

```
print(cr1 == cr2)
print(cr3 == cr2)
```

```
True
False
```

Adding two CashRegisters

- Implement ***__add__*** method for class *CashRegister*, so we can add 2 registers using operator **+**.
- According to docstring in file *cash_register_special.py*, we are adding two cash registers by summing up cash amount in all their respective denominations
- Note that when we use *a* + *b*, the result is a new object of the same type

How to see the results of our hard work with `dir()`

- **`dir()`**: provides a listing of all the attributes and methods of a new object, including the ones inherited from the class *object*

```
cr1 = CashRegister(2, 0, 0, 0, 0)
print (dir(cr1))
```

```
['__add__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'add', 'fives', 'get_total', 'ones',
'remove', 'tens', 'twenties', 'twos']
```

Summary

- *Object* - a collection of attributes (data) and methods (functions)
- A *class* statement provides a **blueprint for creating objects**.
- In Python all data are objects. An object's type corresponds to its class
- Operators (+, ==, >, <) can be overloaded so that the operation performed depends on the class of the operands

What to overload to make your new type behave properly

Which method to overload	Goal	Operator	Returns
__**lt**__(self, other) __**le**__ (self, other) __**gt**__ (self, other) __**ge**__ (self, other) __**eq**__ (self, other) __**ne**__ (self, other)	Comparison, sorting	self < other self **<=** other self **>** other self **>=** other self **==** other self **!=** other	Boolean
__**add**__ (self, other) __**sub**__ (self, other) __**mul**__ (self, other) __**div**__ (self, other)	Numerical operations	self **+** other self **-** other self ***** other self **/** other	Returns new object
__**iadd**__ (self, other) __**isub**__ (self, other)	In-place modifiers	self **+=** other self **-=** other	Replaces current object with a new one