Classes

Thomas Schwarz, SJ

Address Class

- How to generate addresses
  - Each country has its own way of generating addresses
  - An address consists of
    - an optional modifier (apartment, floor, neighborhood)
    - a street
    - a street number
    - a city
    - a state (in most of the Americas)
    - a country

Address Class

- To deal with optional arguments:
  - Use a default argument of none

```
def __init__(self, country,  city, street, number,
             postal,    state, apartment = None):
```

Aside: How to deal with long lines in Python

- Python statements ideally fit in a single line
- In fact, if you want to write poorly readable code, you can put more than one statement in a line and separate with a semi-colon (;)
- Python still allows to use a single forward slash as a continuation marker
- But this is not very readable
- Put expressions into parentheses (unless they already come with parentheses)
- Python interpreter will interpret correctly

The purpose of str and repr

- The dunder methods __str__ and __repr__ seem to do the same thing,
  - But:
    - __str__ is called by print with priority over __repr__
      - This is how you want your output be displayed
    - __repr__ should represent the internal structure of your class instances

Addresses

- We can use __repr__ to just give us the internal makeup of an Address instance

```
def __repr__(self):
       return "apartment: {0}\nstreet: {1}\nnumber: {2}\ncity: {3}\npostal: {4}\nstate: {5}, \ncountry: {6}".format(
           self.apartment, self.street, self.number, self.city, self.postal, self.state, self.country)
```

Addresses

- But for __str__, we will let the country code determine what to do.
- The code is ugly, but that is the price for internationalization
- And we have not even discussed how to be able to use non-English keyboard letters in Python

Self Test

- Open up the file address.py
    - Edit the __str__ dunder method to allow for Indian addresses

Addresses

- When we use str(my_address) on an Address object, we get the result of __str__
- When we use repr(my_address), we get the result of __repr__

Instances can be fields of classes

- When we model processes (such as business processes), we will build up our entities from simpler entities
  - We can have a has-a relationship
  - For example, each person has an address
    - (With many sad exceptions: some have none, some have more than one)

Modular programming

- Remember modules:
  - They are just py-files
  - They are imported using import statements
  - The form of the import statements determines how the names are being resolved
    - `import address`
      - imports the module, names are prefixed with “`address.`”
    - `from address import *`
      - Not recommended, just use names without prefix
    - `from address import Address`
      - Just as before, but only imports the class Address

Client Example

- Clients have a name and an address

```
import address

class Client:
    def __init__(self, name, address):
        self.name = name
        self.address = address
    def __str__(self):
        return "{}\n{}".format(self.name, str(self.address))
    def __repr__(self):
        return "Name: {}\n {}".format(self.name, repr(self.address))

if __name__=="__main__":
    address4 = address.Address("Canada", "Ottawa", "Wellington Street",
                       80, "ON K1A 0A2", "Ontario",
                       "Office of the Prime Minister")
    trudy = Client("The Honorable Justin Trudeau", address4)
    print(trudy)
```

Doc Strings

- Classes are reusable
  - No need to reinvent a working name class
  - But need to provide documentation
  - In Python:
    - This is done primarily with the so-called doc string
      - Right after the definition of a class or function
      - Included between triple quotes

Doc Strings

- The contents are made available to the help function

Example

- A simple checking account class

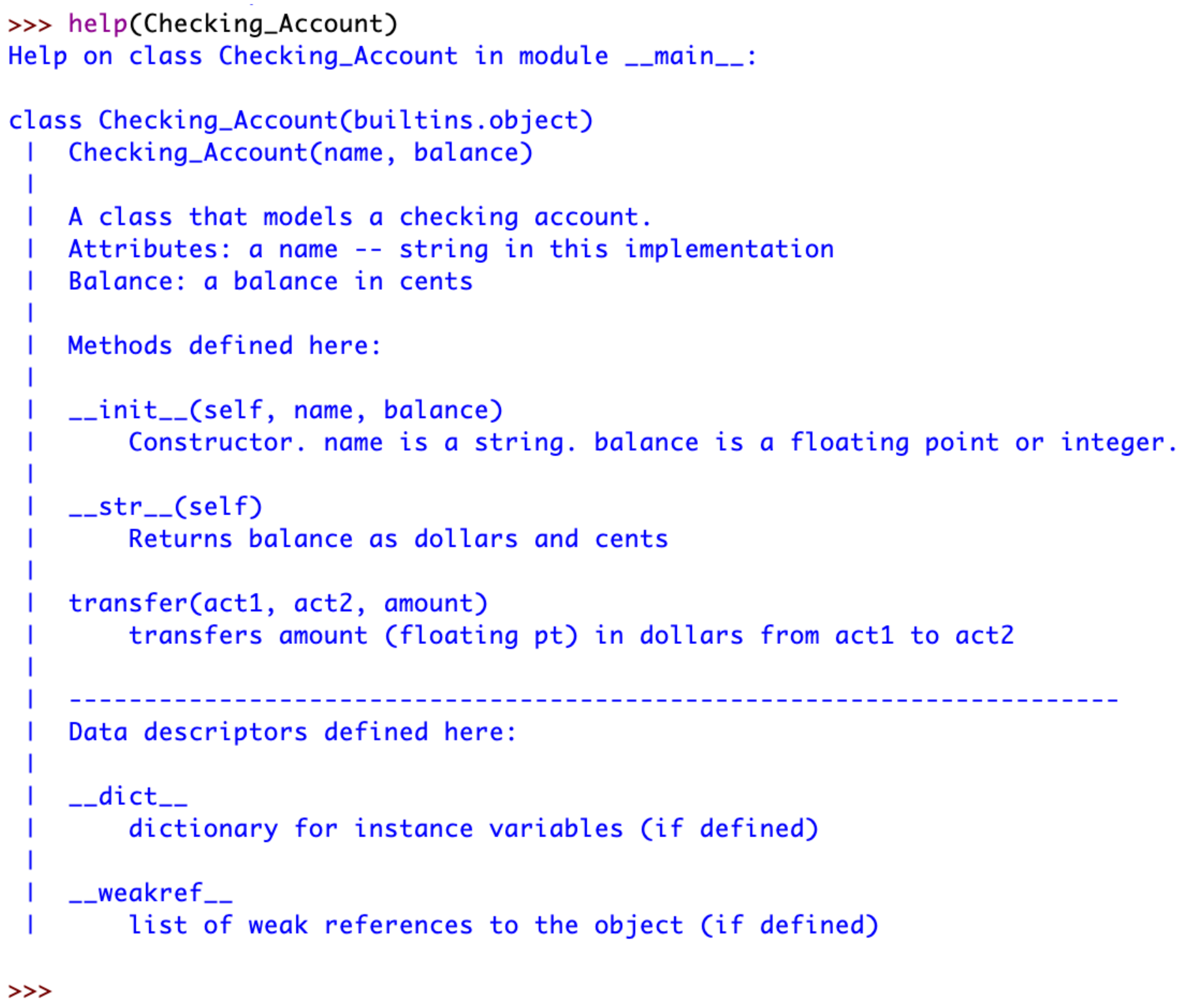
```
class Checking_Account:
    """A class that models a checking account.
        Attributes: a name -- string in this implementation
        Balance: a balance in cents
    """
    def __init__(self, name, balance):
        """Constructor. name is a string. balance is a floating point or integer."""
        self.name = name
        self.balance = round(balance*100)
    def __str__(self):
        """Returns balance as dollars and cents"""
        return "Account for {} with balance US${:d}.{:02d}".format(
            self.name,
            self.balance//100,
            self.balance%100)
    def transfer(act1, act2, amount):
        """transfers amount (floating pt) in dollars from act1 to act2"""
        amount = round(amount*100)
        act1.balance -= amount
        act2.balance += amount
```

Example

```
if __name__ == "__main__":
    a1 = Checking_Account("Thomas Schwarz", 1543.285)
    a2 = Checking_Account("Joseph Cuelho", 1009)
    print(a1)
    print(a2)
    print("Transferring")
    Checking_Account.transfer(a1, a2, 500.01)
    print(a1)
    print(a2)
```

Example

- This is the result of typing help(Checking_Account)



```
>>> help(Checking_Account)
Help on class Checking_Account in module __main__:

class Checking_Account(builtins.object)
 |  Checking_Account(name, balance)
 |  
 |  A class that models a checking account.
 |  Attributes: a name -- string in this implementation
 |  Balance: a balance in cents
 |  
 |  Methods defined here:
 |  
 |  __init__(self, name, balance)
 |      Constructor. name is a string. balance is a floating point or integer.
 |  
 |  __str__(self)
 |      Returns balance as dollars and cents
 |  
 |  transfer(act1, act2, amount)
 |      transfers amount (floating pt) in dollars from act1 to act2
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)

>>>
```

Example

- As you can see, Python has automatically created a help file from the information you provided.

Tricks with Currency Amounts

- Currency is usually a decimal number with exactly two digits precision.
    - Could use the decimal - class
    - Could use third party classes
    - We build our own
- Idea: Present currency as multiples of cents.

```
class Checking_Account:
    """A class that models a checking account.
        Attributes: a name -- string in this implementation
        Balance: a balance in cents
    """
    def __init__(self, name, balance):
        """Constructor. name is a string. balance is a
           floating point or integer.
        “""
        self.name = name
        self.balance = round(balance*100)
```

Tricks with Currency Accounts

- To print out currencies, we break the cents apart into the dollars (displayed normally) and the cents amount proper.
  - The format mini-language allow us to print out amounts with leading 0.
  - Just stick a 0 in front of the width field

```
def __str__(self):
    """Returns balance as dollars and cents"""
    return "Account for {} with balance US${:d}.{:02d}".format(
        self.name,
        self.balance//100,
        self.balance%100)
```

Specify leading zero in the format mini-language

Self Test

- Modify the __str__ function so that a negative amount is written in the form
    - `-US$103.05`

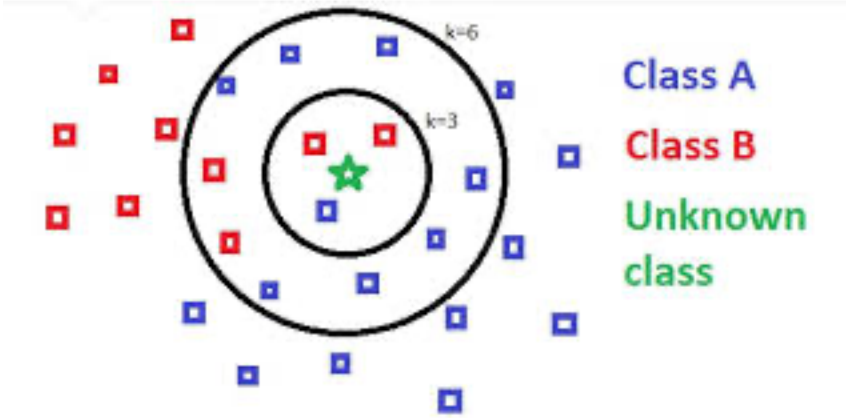
Solution

- Just make a case distinction, but make sure that you do not change the field

```
    def __str__(self):
        """Returns balance as dollars and cents"""
        if self.balance >= 0:
            return "Account for {} with balance US${:d}.{:02d}".format(
                self.name,
                self.balance//100,
                self.balance%100)
        else:
            balance = -self.balance
            return "Account for {} with balance -US${:d}.{:02d}".format(
                self.name,
                balance//100,
                balance%100)
```

K Nearest Neighbor

- A simple classification system
  - Classify an unknown category by looking at the *k* nearest neighbors



K Nearest Neighbor

- How do we define near-ness
  - One possibility: Euclidean distance
  - Data points with numerical values $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$: $\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2}$

k Nearest Neighbor

- Usually need to normalize values
  - Otherwise dimension will matter
    - (100000, 1) and (1000010, 5) are almost equally distant from (0,6)
  - Normalize: $x \mapsto \dfrac{x - \min}{\max - \min}$
  - Now all coordinates are between 0 and 1

k Nearest Neighbor

- Other distances are possible
  - Angle between the two points $\arccos(\frac{\mathbf{x} \cdot \mathbf{y}}{|\mathbf{x}||\mathbf{y}|})$
  - Weighted euclidean distance
  - Manhattan distance
  - …

k Nearest Neighbor

- Parameter *k* has an influence on accuracy:
  - Choose odd *k* to deal with ties when we have only two categories

knn Implementation

- Want to do something more generic
    - Assume a csv file:
        - First column might be an index
        - Then observable values
        - Finally category
- Want to normalize:
    - Need to find maximum and minimum for each coordinate
        - This goes into class variables

knn Implementation

- Create a class for data: Cat_Data

```
class Cat_Data():
    nr_cols = 0
    mins = []
    maxs = []
```

knn Implementation

- Whenever an object is created, we update the three class variables

```
def __init__(self, data, cat):
        self.values = data
        self.cat = cat
        if len(self.values) > Cat_Data.nr_cols:
            Cat_Data.mins.extend(data[Cat_Data.nr_cols: ])
            Cat_Data.maxs.extend(data[Cat_Data.nr_cols: ])
            Cat_Data.nr_cols = len(self.values)
        for i, val in enumerate(self.values):
            if val < Cat_Data.mins[i]:
                Cat_Data.mins[i] = val
            if val > Cat_Data.maxs[i]:
                Cat_Data.maxs[i] = val
```

knn Implementation

- Need to create string dunder

```
def __str__(self):
        retVal = []
        for val in self.values:
            retVal.append(str(val))
        retVal.append('cat: ' + str(self.cat))
        return ', '.join(retVal)
```

knn Implementation

- The repr dunder is mainly the same

```
def __repr__(self):
        retVal = ['Cat_Data']
        for val in self.values:
            retVal.append(str(val))
        retVal.append('cat: ' + str(self.cat))
        return ', '.join(retVal)
```

knn Implementation

- Create a class method 'load'
  - Takes file name and as optional parameter, whether the first column is an index column

```
def load(file_name, index = True):
        lista = []
        with open(file_name) as infile:
            infile.readline() # remove first line
            for line in infile:
                contents = line.strip().split(',')
                data = []
                if index:
                    contents = contents[1:]
                for val in contents[:-1]:
                    data.append(float(val))
                cat = contents[-1]
                lista.append(Cat_Data(data, cat))
        return lista
```

knn Implementation

- To normalize, need to know the value and the coordinate

```
def normalize(val, i):
        return (val-Cat_Data.mins[i])/(Cat_Data.maxs[i]-
Cat_Data.mins[i])
```

knn Implementation

- Distance between points is the Euclidean distance between normalized data points

```
def distance(self, other):
        yog = 0
        for i in range(min(len(self.values),
len(other.values))):
            yog += (Cat_Data.normalize(self.values[i], i)-
Cat_Data.normalize(other.values[i], i))**2
        return math.sqrt(yog)
```

knn Implementation

- Now can write a classifier
    - Needs to find the nearest *k* elements
        - We can speed this up by limiting the number of elements that we need to look at
            - E.g. using a kd-tree
        - But here, we just order all data points by their distance
        - Use Counter and sort with a key function

knn Implementation

```
def classify(element, lista, k=5):
    distances = [ (el, element.distance(el)) for el in lista ]
    distances.sort(key = lambda x:  x[1])
    votes = Counter( [ x[0].cat for x in distances[:k]] )
    return votes.most_common(1)[0][0]
```