Object oriented programming

IC152 Feb 2021

OOP

- OOP is a programming paradigm that structres program so that properties and behaviour are bundled together, into objects.
- Another paradigm: procedural programming
  Divide a program into functions (this is mostly what we have been using so far.)

- Simple data structures like integers, strings (*)
- More complex data: eg. employee information
  - Name, EmpId, DoB, Location, Dept, DoJ, reportingTo

Rajat Kumar
1023345
09-10-1980
Delhi
Sales
12-3-2019
3445235

Meena Singh
3445235
09-10-1971
Mumbai
Sales
12-3-2001
5662234

How to represent this type of data?

We can use a collection of lists, like this:

E1 = [‘Rajat Kumar’, ’1023456’, ’03-10-1980’, ’Sales’, ‘4556234’]
E2 = [‘Meena Singh’,’4556234’,’14-05-1971’,’Sales’,’76654556’]

and so on

Problems with this include:

- Keeping track of indices
- Hard to maintain
- This is not the use of a list

Classes

- Classes are used to create user-defined data structures
- A class is a template for creating an instance, called an object
- Classes usually have attributes and methods

https://realpython.com/python3-object-oriented-programming/

```
 8
 9  class Dog:
10
11      # Class attribute
12      species = 'Canis familiaris'
13
14      def __init__(self, name, age):
15          self.name = name
16          self.age = age
```

Class attributes

Constructor method

Instance attributes

```
In [10]: t = Dog('Teddy',5)

In [11]: r = Dog('Russo',3)

In [12]: type(t)
Out[12]: __main__.Dog

In [13]: t.name
Out[13]: 'Teddy'

In [14]: r.age
Out[14]: 3

In [15]: r.species
Out[15]: 'Canis familiaris'

In [16]: t.species
Out[16]: 'Canis familiaris'
```

```
In [19]: t.age
Out[19]: 5

In [20]: t.age=7

In [21]: t.species = 'Canis lupus'

In [22]: t
Out[22]: <__main__.Dog at 0x7fdca0070760>

In [23]: t.species
Out[23]: 'Canis lupus'
```

Attributes can be updated (they are mutable)

```python
class Dog:

    # Class attribute
    species = 'Canis familiaris'

    def __init__(self, name, age):
        self.name = name
        self.age = age




t = Dog('Teddy',5)
r = Dog('Russo',3)
t.species = 'Canis lupus'
```

```
In [26]: t.species
Out[26]: 'Canis lupus'

In [27]: r.species
Out[27]: 'Canis familiaris'
```

```python
class Dog:
    tricks = []     # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)


d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks
```

```
In [32]: d.tricks
Out[32]: ['roll over', 'play dead']
```

Careful while using mutable objects as class attributes!

Better class design ⟶

```python
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)


d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
```

```python

class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f'{self.name} is {self.age} years old'

    # Another instance method
    def speak(self, sound):
        return f'{self.name} says {sound}'

    def __str__(self):
        return f"{self.name} is a dog, and is {self.age} years old"



miles = Dog('Miles', 4)
print(miles.description())
print(miles.speak('Woof Woof'))
print(miles.speak('Bow Wow'))
print(miles)

```

`Fstring` string formatting

Instance method

Dunder method

Instance methods can be used, for example, to check for validity of input

```
Miles is 4 years old
Miles says Woof Woof
Miles says Bow Wow
Miles is a dog, and is 4 years old
```

Tip: use the built-in `dir()` function to list attributes and methods of any object

Inheritance

(Usually) more specialized

Child classes can override or extend attributes from the parent class

Hair colour from mother
But you can override it

Add an attribute that your parents don't have (eg. learning a new language)

```python
class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f'{self.name} is {self.age} years old'

    # Another instance method
    def speak(self, sound):
        return f'{self.name} says {sound}'

    def __str__(self):
        return f'{self.name} is a dog, and is {self.age} years old'



class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass


miles = JackRussellTerrier('Miles', 4)
buddy = Dachshund('Buddy', 9)
jack = Bulldog('Jack', 3)
jim = Bulldog('Jim', 5)

print(buddy)
print(f'Jim is {jim.age} years old.')

```

Parent class

```
Buddy is a dog, and is 9 years old
Jim is 5 years old.

In [53]: jim.speak('woof')
Out[53]: 'Jim says woof'

In [54]: jack.speak('roar')
Out[54]: 'Jack says roar'
```

Empty class definition

```python
class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f'{self.name} is {self.age} years old'

    # Another instance method
    def speak(self, sound):
        return f'{self.name} says {sound}'

    def __str__(self):
        return f'{self.name} is a dog, and is {self.age} years old'


class JackRussellTerrier(Dog):
    def speak(self, sound='Arf'):
        return f'{self.name} says {sound}'

class Dachshund(Dog):
    def speak(self, sound='Yap'):
        return f'{self.name} says {sound}'

class Bulldog(Dog):
    def speak(self, sound='Bow'):
        return f'{self.name} says {sound}'


miles = JackRussellTerrier('Miles', 4)
buddy = Dachshund('Buddy', 9)
jack = Bulldog('Jack', 3)
jim = Bulldog('Jim', 5)

print(jim.speak())
print(buddy.speak())
```

```
Jim says Bow
Buddy says Yap

In [57]: buddy.speak('growl')
Out[57]: 'Buddy says growl'
```

Overriding the method of the base class

`isinstance()` and `issubclass()` built-in functions

HW

```
 8
 9  class Employee:
10      pass
11
12  john = Employee()  # Create an empty employee record
13
14  # Fill the fields of the record
15  john.name = 'John Doe'
16  john.dept = 'computer lab'
17  john.salary = 1000
18
19  mary = Employee()
20
```

An empty class can be used to bundle a few names attributes

```
In [64]: john.name
Out[64]: 'John Doe'

In [65]: mary.name
Traceback (most recent call last):

  File "<ipython-input-65-e5b69b3c3be9>", line 1, in <module>
    mary.name

AttributeError: 'Employee' object has no attribute 'name'
```

```python

from collections import Counter

x = [2, 2, 0, 0, 5, 8, 3, 4, 1, 0, 0, 7, 1, 7, 1, 5, 4, 0, 4,\
0, 1, 8, 9, 7, 0, 1, 7, 2, 5, 5, 4, 3, 3, 0, 0, 2, 5, 1, 3, 0,\
1, 0, 2, 4, 5, 0, 5, 7, 5, 1]

c = Counter(x)
print(c.keys())
print(c.values())

```

```
dict_keys([2, 0, 5, 8, 3, 4, 1, 7, 9])
dict_values([5, 12, 8, 2, 4, 5, 8, 5, 1])

In [113]: c[2]
Out[113]: 5

In [114]: c[6]
Out[114]: 0
```

Calend × | collections — Contai × | Built-in Types — Pyth × | collections — Contai × | Object-Oriented Prog × | 9. Class

docs.python.org/3/library/collections.html#collections.Counter

Imported | Google Calend... | padman | IC152-18 | IC152-21 | IC152-19 | ic152-2021 | Sl... | MoodleCloud

```
('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.**Counter**([*iterable-or-mapping*])

A `Counter` is a `dict` subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The `Counter` class is similar to bags or multisets in other languages.

Elements are counted from an *iterable* or initialized from another *mapping* (or counter):

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a `KeyError`:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

Counting the number of occurrences

```
 8
 9 # Skeleton code for processing transactions
10
11
12 def processCashTrans(t):
13     try:
14         ct = CashTrans(t)
15         ct.verify()
16         ct.update()
17     except CashTransError:
18         # do something
19
20 def processChequeTrans(t):
21     try:
22         cht = ChequeTrans(t)
23         cht.verify()
24         cht.update()
25     except ChequeTransError:
26         # do something
27
28 def processWalletTrans(t):
29     try:
30         wt = WalletTrans(t)
31         wt.verify()
32         wt.update()
33     except WalletTransError:
34         # do something
35
36
37
38 lst_trans = readTransactions(inputFile)
39 for t in lst_trans:
40     if t.type == 'CQ':
41         processChequeTrans(t)
42     elif t.type == 'CH':
43         processCashTrans(t)
44     elif t.type == 'WL':
45         processWalletTrans(t)
46
47
```

Create CashTrans object

Operations specific to cash

Create ChequeTrans object

Operations specific to cheques

And so on

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon May  3 16:01:47 2021

@author: paddy
"""

class Point:
    """A Point instance models a 2D point with x and y coordinates"""

    def __init__(self, x = 0, y = 0):
        """Initializer, which creates the instance variables x and y with default of (0, 0)"""
        self.x = x
        self.y = y

    def __str__(self):
        """Return a descriptive string for this instance"""
        return '({}, {})'.format(self.x, self.y)

    def __repr__(self):
        """Return a command string to re-create this instance"""
        return 'Point(x={}, y={})'.format(self.x, self.y)

    def __add__(self, right):
        """Override the '+' operator: create and return a new instance"""
        p = Point(self.x + right.x, self.y + right.y)
        return p

    def __mul__(self, factor):
        """Override the '*' operator: modify and return this instance"""
        self.x *= factor
        self.y *= factor
        return self

# Test

p1 = Point()
print(p1)      # (0.00, 0.00)
p1.x = 5
p1.y = 6
print(p1)      # (5.00, 6.00)
p2 = Point(3, 4)
print(p2)      # (3.00, 4.00)
print(p1 + p2) # (8.00, 10.00) Same as p1.__add__(p2)
print(p1)      # (5.00, 6.00) No change
print(p2 * 3)  # (9.00, 12.00) Same as p1.__mul__(p2)
print(p2)      # (9.00, 12.00) Changed
```

Overriding __add__ allows us to add objects p1 and p2

https://www3.ntu.edu.sg/home/ehchua/programming/webprogramming/Python1a_OOP.html

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue May  4 15:23:13 2021

@author: paddy
"""

class Time:

    def __init__(self,h=0,m=0,s=0):
        self.hrs=h
        self.min=m
        self.sec=s

    def printTime(self):
        return(str(self.hrs)+':'+str(self.min)+':'+str(self.sec))

    def __str__(self):
        return self.printTime()

    def increment(self,sec):
        self.sec = self.sec + sec
        while self.sec >= 60:
            self.sec = self.sec - 60
            self.min = self.min + 1
        while self.min >= 60:
            self.min = self.min - 60
            self.hrs = self.hrs + 1

    """ Returns 1 if time2 is after time1 (self), else returns 0"""
    def after(self,time2):
        pass

    """addition"""
    def __add__(self,time2):
        # convert everything into seconds
        if time2.hrs > 0:
            time2.sec = time2.sec + time2.hrs * 3600
        if time2.min > 0:
            time2.sec = time2.sec + time2.min * 60
        self.increment(time2.sec)


# main
t1 = Time(10,0,0)
print(t1)
t2 = Time(h=1,m=1,s=0)
t1 + t2
print(t1)
```

HW

```python
import numpy as np


class Histogram:
    def __init__(self,n):
        self.freq = np.zeros(shape=(1,n+1))
        self.maxval = n

    def addDataPoint(self,x):
        if x>self.maxval:
            raise ValueError
        self.freq[0,x] = self.freq[0,x] + 1

    def __str__(self):
        return str(self.freq)


h = Histogram(10)
rng = np.random.default_rng()
rints = rng.integers(low=0, high=10, size=20)
print(rints)
for x in rints:
    h.addDataPoint(x)
print('Printing histogram:')
print(h)

```

```
feb2021/ic152/lectures/lec26_oop/code')
[6 5 6 0 6 7 0 1 7 4 3 4 1 8 7 1 5 1 6 8]
Printing histogram:
[[2. 4. 0. 1. 2. 2. 4. 3. 2. 0. 0.]]

In [78]: h1 = Histogram(10)

In [79]: h1.addDataPoint(3)

In [80]: h1.addDataPoint(7)

In [81]: h1.addDataPoint(3)

In [82]: h1.addDataPoint(4)

In [83]: print(h1)
[[0. 0. 0. 2. 1. 0. 0. 1. 0. 0. 0.]]
```