Recap and language details for Python exercises

Lab 5 - OOP: inheritance. Exceptions

Inheritance

Inheritance

- Inheritance provides a way to share functionality between classes.
  - Imagine several classes, `Cat`, `Dog`, `Rabbit` and so on. Although they may differ in some ways (only `Dog` might have the method `bark`), they are likely to be similar in others (all having the attributes `color` and `name`).
  - This similarity can be expressed by making them all inherit from a superclass `Animal`, which contains the shared functionalities.
- A class that inherits from another class is called a **subclass**.
- A class that is inherited from is called a **superclass**.
- To inherit a class from another class, put the superclass name in parentheses after the class name.
- If a subclass has attributes or methos with the same name of ones in the superclass, subclass overrides them.
- One class can inherit from another, and that class can inherit from a third class.
- Private attributes (starting with `__`) are not visible in subclasses.

Inheritance: Example

```
1  class Animal: # superclass
2    def __init__(self, name, color): # shared
       functionality
3      self.name = name
4      self.color = color
5
6  class Dog(Animal): # subclass
7    def bark(self):
8      print('Woof!')
9
10 class Cat(Animal): # subclass
11   def purr(self):
12     print('Purr...')
13
14 fido = Dog('Fido', 'brown')
15 fido.bark()
16 ginger = Cat('Ginger', 'red')
17 ginger.purr()
```

Inheritance: Example (2)

```
1  class Animal: # superclass
2    def __init__(self, name, color):
3      self.name = name
4      self.color = color
5
6  class Dog(Animal): # subclass and superclass
7    def bark(self):
8      print('Woof!')
9
10 class Wolf(Dog): # subclass
11   def bark(self): # method overriding
12     print('Grrr...')
```

Inheritance: dynamic selection of methods

```
>>> fido = Dog('Fido', 'brown')
>>> fido.bark()
Woof!
>>> sharp = Wolf('Sharp Tooth', 'grey')
>>> sharp.bark()
Grrr...
```

Is an object instance of a class/superclass?

We can use the special function **isinstance** to determine if an object is an instance of a certain class and/or its superclasses.

On the other hand, **type** returns only the specific class (the type)

```
>>> sharp = Wolf('Sharp Tooth', 'grey')
>>> type(sharp)
<class '__main__.Wolf'>
>>> isinstance(sharp,Wolf)
True
>>> isinstance(sharp,Dog)
True
>>> type(sharp) is Dog
False
>>> isinstance(sharp,Animal)
True
>>> isinstance(sharp,Cat)
False
```

The super() function i

We can use the **super**`()` function to "read" the `self` object as if it were an instance of its superclass.

In the following example, within the class `Cat`, **super**`().__init__()` refers to the `__init__` of `Cat` (which in turn inherits the `__init__` of `Animal`).

The call **super**`().meow()` refers to the `meow()` of `Cat`.

The super() function ii

```
class Animal: # superclass
  def __init__(self, name, color):
    self.name = name
    self.color = color

class Cat(Animal):
  def meow(self):
    print('Miao!')

class Siamese_Cat(Cat):
    def __init__(self, name):
        super().__init__(name, "white-cream-black")
        self.pedigree = True
    def meow(self):
        for i in range(3):
            super().meow()
```

Exceptions

Exceptions

- Are *raised* when a run-time/dynamic error happens
- Can be caught with the **try** . . . **except** construct
- Both pre-existing and user-defined exceptions can be raised with the **raise** command
- User-defined exceptions are subclasses of `Exception` class

Try... except... else... finally

```
1  try:
2      <block> #try block is executed first
3  except Exc1:
4      <block> #executed if try block raises Exc1
5  except (Exc2, Exc3):
6      <block> #if the try raises one of Exc2 or Exc3
7  except Exc4 as name: #if try raises Exc4;
8      <block> #name bound to instance of Exc4 in order
       to read its value (e.g. descriptive string)
9  except:
10     <block> #if the try raises any other exceptions
11 else:
12     <block> #if try block does not raise exceptions
13 finally:
14     <block> # executed anyway before leaving the try
```

else and **finally** blocks are optional. Arbitrary the number and type of **except** blocks (and thus optional the generic **except** block).

Rasing exeptions

- Exceptions are raised by the Python machine when something happens (e.g. `1/0` will raise the `ZeroDivisionError` exception)
- We can deliberately raise exeptions with the **raise** command
- We can raise both pre-defined and user-defined exceptions

Better to ask forgiveness than permission?

Python programmers prefer the EAFP style: *"Easier to ask for forgiveness than permission"*.

Often, programmers of other languages prefer an LBYL approach: *"Look before you leap"*.

Better to ask forgiveness than permission?

So in Python it is preferable to write...

```
1 D = {'apples':3, 'peaches':4}
2 try:
3     print(D['bananas'])
4 except KeyError:
5     print("There are no bananas")
```

instead of ...

```
1 D = {'apples':3, 'peaches':4}
2 if 'bananas' in D:
3     print(D['bananas'])
4 else:
5     print("There are no bananas")
```

We never encouraged this style, but at least in today's lab try to get into this mindset: I try, and if it doesn't work, I manage.

A comprehension exercise

What will the following code print?

```
1  class MyException(Exception): pass
2
3  def f(x):
4      if x < 0:
5          raise MyException
6      if type(x) != int:
7          raise TypeError
8      print(x)
9
10 for v in [-3, 2.5, 77]:
11     try:
12         f(v)
13     except MyException:
14         print("Insert a positive number")
15     except TypeError:
16         print("Insert an integer")
```