Classes and Object-Oriented Programming in Python

Here I demonstrate several key features of classes and object-oriented programming in Python. These include the following

- class inheritance
- overriding magic methods
- property setters and getters (using `@property` and `@<attribute>.setter` decorators)
- composition

In [1]:
```
import datetime
import time
```

I will start by defining a `Person` class.

In [2]:
```
class Person(object):

    def __init__(self, name, surname, gender, birthdate, **kwargs):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate
        if 'male' == gender.lower() or 'boy' == gender.lower():
            self.gender = 'male'
        else:
            self.gender = 'female'
        # accomodate other input information
        for key, val in kwargs.items():
            self.__dict__[key] = val

    @property
    def fullname(self):
        return '{} {}'.format(self.name, self.surname)

    @property
    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birt
            age -= 1
        return age

    # override some magic methods

    def __str__(self):
        critical_keys = ['name', 'surname', 'gender', 'birthdate', 'fullname
        about = '\n{} is a {}-y-o {}.\n'.format(self.fullname, self.age,
                                                 self.gender)
        additional_keys = list(set(self.__dict__.keys()) - set(critical_keys
        if len(additional_keys) > 0:
            about += '--Additional info--\n'
            for key in additional_keys:
                about += '{}: {}\n'.format(key, self.__dict__[key])
        return about

    # setup comparison based on name alphabatization
    def __eq__(self, other): # does self == other?
        return self.name == other.name and self.surname == other.surname

    def __gt__(self, other): # is self > other?
        if self.surname == other.surname:
            return self.name > other.name
        return self.surname > other.surname

    # now we can define all the other methods in terms of the first two
    def __ne__(self, other): # does self != other?
        return not self == other # this calls self.__eq__(other)

    def __le__(self, other): # is self <= other?
        return not self > other # this calls self.__gt__(other)
```

```
    def __lt__(self, other): # is self < other?
        return not (self > other or self == other)

    def __ge__(self, other): # is self >= other?
        return not self < other
```

This expects certain critical input values when intantiating a person, their first and last name, gender, and birthdate.

In [3]:
```
dawn = Person('Dawn', 'Joe', 'female', datetime.date(1984, 1, 13))
print(dawn)
```

```
Dawn Joe is a 34-y-o female.
```

For demonstration purposes, I set this up to accept additional keyword arguments as well. This should be done with care as relying on these additional attributes can lead to problems if they are not populated.

In [4]:
```
jon = Person('Jon', 'Doe', 'male', datetime.date(1983, 8, 21),
             email='jon.doe@email.com', address='123 Redwood Ct',
             cell='249.298.6690', hair='red')
print(jon)
```

```
Jon Doe is a 35-y-o male.
--Additional info--
hair: red
cell: 249.298.6690
address: 123 Redwood Ct
email: jon.doe@email.com
```

Now lets define a `Child` class. It will inherit from the `Person` class, with one added property, `nap_time`.

In [5]:
```
class Child(Person):

    @property
    def nap_time(self):
        if self.age < 1:
            return [9, 1]
        elif self.age < 5:
            return [1]
        else:
            return []

    def __str__(self):
        critical_keys = ['name', 'surname', 'gender', 'birthdate', 'fullname
        about = '{} is a {}-y-o {},\n'.format(self.fullname, self.age,
                                               self.gender)
        if len(self.nap_time) > 1:
            about += 'and takes naps at {} and {} o-clock.\n'.format(*self.n
        elif len(self.nap_time) > 0:
            about += 'with a nap time at {} o-clock.\n'.format(*self.nap_tim
        else:
            about += 'and is too old for naps.\n'

        additional_keys = list(set(self.__dict__.keys()) - set(critical_keys
        if len(additional_keys) > 0:
            about += '--Additional info--\n'
            for key in additional_keys:
                about += '{}: {}\n'.format(key, self.__dict__[key])
        return about
```

In [6]:
```
sussy = Child('Sussy', 'Doe', 'female', datetime.date(2011, 7, 22))
print(sussy)
```

```
Sussy Doe is a 7-y-o female,
and is too old for naps.
```

In [7]:
```
johnny = Child('Johnny', 'Doe', 'male', datetime.date(2015, 3, 1),
               blankie='blue', hair='red', toy='green ball')
print(johnny)
```

```
Johnny Doe is a 3-y-o male,
with a nap time at 1 o-clock.
--Additional info--
hair: red
blankie: blue
toy: green ball
```

I have already been using composition, by providing a datetime object for the `birthdate` input value but I will go one step further. Now I will define a basic family class.

In [8]:
```
class Family(object):

    def __init__(self, mommy, daddy, *kids):
        self.mommy = mommy
        self.daddy = daddy
        self.kids = list(kids)
        self.number_of_kids = len(self.kids)

    def __str__(self):
        about = ('\nThe {} family is made up of {}, {}, \n'
                 'and their {} kids: \n'.format(self.daddy.surname,
                                                self.daddy.name,
                                                self.mommy.name,
                                                self.number_of_kids))
        for kid in self.kids:
            about += '{name}\n'.format(name=kid.name)

        return about
```

I will use this to define a family from the four people I have already instantiated, Jon, Dawn, Sussy, and Johnny.

In [9]:
```
simple_family = Family(dawn, jon, johnny, sussy)
print(simple_family)
```

```
The Doe family is made up of Jon, Dawn,
and their 2 kids:
Johnny
Sussy
```

Now I will define a family that can add grow using the `add_child` method. I will implement this two different ways to illustrate the difference between lazy and eager calculations.

In [10]:
```
class LazyFamily(Family):

    def __init__(self, mommy, daddy, *kids):
        self.mommy = mommy
        self.daddy = daddy
        self.kids = list(kids)

    @property
    def family_size(self):
        time.sleep(0.01)  # mimic a long calculation
        return 2 + self.number_of_kids

    @property
    def number_of_kids(self):
        time.sleep(0.01)  # mimic a long calculation
        return len(self.kids)

    def add_child(self, child):
        self.kids.append(child)
```

In [11]:
```
alicia = Child('Alicia', 'Doe', 'female', datetime.date(2017, 7, 20))
family_1 = LazyFamily(dawn, jon, johnny, sussy)
print('before: {}'.format(family_1))

family_1.add_child(alicia)
print('after: {}'.format(family_1))
```

```
before:
The Doe family is made up of Jon, Dawn,
and their 2 kids:
Johnny
Sussy

after:
The Doe family is made up of Jon, Dawn,
and their 3 kids:
Johnny
Sussy
Alicia
```

In [12]:
```
class EagerFamily(Family):

    def __init__(self, mommy, daddy, *kids):
        self.mommy = mommy
        self.daddy = daddy
        self.kids = list(kids)
        self._number_of_kids = len(self.kids)
        self._family_size = 2 + self.number_of_kids

    @property
    def number_of_kids(self):
        return self._number_of_kids
    @number_of_kids.setter
    def number_of_kids(self, val):
        time.sleep(0.01)  # mimic a long calculation
        self._number_of_kids = val

    @property
    def family_size(self):
        return self._family_size
    @family_size.setter
    def family_size(self, val):
        time.sleep(0.01)  # mimic a long calculation
        self._family_size = val

    def add_child(self, child):
        self.kids.append(child)
        self._number_of_kids = len(self.kids)
        self._family_size = 2 + self.number_of_kids
```

In [13]:
```
family_2 = EagerFamily(dawn, jon, johnny, sussy)
print('before: {}'.format(family_2))

rosy = Child('Rosy', 'Doe', 'female', datetime.date(2017, 1, 19))
family_2.add_child(rosy)
print('after: {}'.format(family_2))
```

```
before:
The Doe family is made up of Jon, Dawn,
and their 2 kids:
Johnny
Sussy

after:
The Doe family is made up of Jon, Dawn,
and their 3 kids:
Johnny
Sussy
Rosy
```

On the surface, these two different Family definitions seems to perform the same function. They differ in how they are calculating some of the properties, particularly `family_size` and `number_of_kids` . In the `LazyFamily` , nothing is calculated until it is asked for. In the `EagerFamily` , the calculations are performed as soon the information is available then cached or stored until needed. In this example, the operations are fairly minimal, so I added a 10 ms `sleep` before the calculation in both family definitions. This provides a clear comparison between the timing results of the two different approaches.

Here I query the `family_size` five times. In the lazy case, this means the computation to get the `family_size` must be performed five times instead of just once.

In [14]:
```
n = 5

t0 = time.time()
for i in range(n):
    lazy_family_size = family_1.family_size
t_lazy = time.time() - t0

t0 = time.time()
for i in range(n):
    eager_family_size = family_2.family_size
t_eager = time.time() - t0

print("lazy family: {}s".format(t_lazy))
print("eager family: {}s".format(t_eager))
x_faster = t_lazy / t_eager
print("eager family was {}x faster than the lazy family".format(x_faster))
```

```
lazy family: 0.11362409591674805s
eager family: 0.0001220703125s
eager family was 930.80859375x faster than the lazy family
```

Notice how large of a difference this was!

Or I can just use the `%timeit` magic function.

In [15]:
```
%timeit family_1.family_size
```

```
22.7 ms ± 933 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [16]:
```
%timeit family_2.family_size
```

```
106 ns ± 3.61 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops eac
h)
```

Note how this confirms that for the eager case, the value only had to be computed once.

In [ ]:
```

```