Recap some of OOP concepts

(see https://docs.python.org/3/tutorial/classes.html)

Note OOP is for a big software project. It may appear awkwardly clumsy for rapid proto-typing, but its benefits will be more apparent in a long development project.

Modularization: class and object, attribute, and method

Encapsulation: public, protected, and private

Python (upto Python 3) does not have real "private" variables in the sense that private variables cannot be accessed from outside an object. But, the convention is to NOT ACCESS "PRIVATE" VARIABLES OUTSIDE AN OBJECT.

Example

```
================================================================
# Define a class
class Trip:
    name = ""     # public attribute
    _duration = 0 # protected attribute
    __budget = 0  # "private" attribute
    route = ""    # public attribute

    # Constructor
    def __init__(self, trip='Trip', length=9, cost=30000,
                 itin='city-rural-wild'):
        print('* Constructor runs.')
        self.name = trip
        self.route = itin
```

```
        self.set_duration(length)
        self.set_budget(cost)

    # Method
    def info(self):
        return self.name + ':' + self.route + ', ' + \
               str(self._duration) + ' day(s), ' + str(self.__budget)

    # Methods to access non-public attributes
    def set_duration(self, days):
        self._duration = days

    def get_duration(self, days):
        return self._duration

    def set_budget(self, baht):
        self.__budget = baht

    def get_budget(self):
        return self.__budget


if __name__ == '__main__':
    t = Trip()     # Instantiate an object
    s = t.info()
    print(s)

    t1 = Trip('Serengeti', 8, 40000, 'camp-safari-tribe')
    s = t1.info()
    print(s)
```

```
    t1.name = 'Wahiba Sands'
    t1.set_duration(7)   # Access a protected variable through a method
    t1.set_budget(35000) # Access a private variable through a method
    t1.route = 'sands-souq-sea-bedouin'
    s = t1.info()
    print(s)
```

==========

Static vs Dynamic

See this example

==========

```
class tree:
    a = 5
    b = []
    c = []

    def __init__(self):
        self.c = []

if __name__ == '__main__':
    t1 = tree()
    t2 = tree()

    print('t1:', t1.a, t1.b, t1.c)
    print('t2:', t2.a, t2.b, t2.c)
    t1.a = 8
    t1.b.append(3)
    t1.c.append(9)
```

```
    print('t1:', t1.a, t1.b, t1.c)
    print('t2:', t2.a, t2.b, t2.c)
```

==========

Make sense of its result

==========

```
t1: 5 [] []
t2: 5 [] []
t1: 8 [3] [9]
t2: 5 [3] []
```

==========

Hint: what makes attribute c different than attribute b. (It's dynamic vs static.)

Inheritance.

==========

```
class MyTrip(Trip):        # Define a class with its parent class
    visa = ""

    def info(self):
        s = super().info() # super() is to refer to its parent
        return s + ': visa=' + self.visa

if __name__ == '__main__':
    # object t1
    t1 = MyTrip('Suzdal', 9, 42000, 'orthodox-slavic-traditional')
    s = t1.info()
    print(s)
```

```
    # test if MyTrip is inherited from Trip
    print(issubclass(MyTrip, Trip))

    # all attributes and methods from parent are inherited.
    t1.name = 'Bukhara'
    t1.route = 'silkroad-crossroad-centralasia'
    t1.set_budget(38000)
    t1.visa = 'Required'
    s = t1.info()
    print(s)

    # object t2
    t2 = MyTrip('Mysore', 8, 35000, 'temple-palace-hillstation')
    s = t2.info()
    print(s)
```

==========

:::Note::: it is not just a new trick. It is a new paradigm. Think of object interaction rather than variables and functions.

Protected vs Private

See this example

==========

```
class Plant:
    a = 5
    _b = 8
    __c = 22

    def info(self):
        print(self.a, self._b, self.__c)
```

```
    def inc(self):
        self.a += 1
        self._b += 1
        self.__c += 1


class Tree(Plant):

    def grow(self):
        self.a *= 2
        self._b *= 2
        # self.__c *= 2   # try uncomment this and run it

if __name__ == '__main__':
    p1 = Plant()
    p1.info()
    p1.inc()
    p1.info()

    t1 = Tree()
    t1.info()
    t1.inc()
    t1.info()
    t1.grow()
    t1.info()
================================================================
```

Don't worry if you don't get all the tricks, just have fun playing with them.

==========

Use `if __name__ == '__main__':` for better code organization and allowing smooth autograding.

P1. Class, object, and attributes. Write a class named `dish` having 2 attributes.

The two attributes are `name` and `description`. `Both are` strings.

Example 1

When the code below is run:

```
==========================================================================
d1 = dish()
print(type(dish))
print(type(d1))
print(d1.name)
print(type(d1.name))
print(d1.description)
print(type(d1.description))
==========================================================================
```

it results

```
==========================================================================
<class 'type'>
<class '__main__.dish'>

<class 'str'>
```

```
<class 'str'>
==========================================================================
```

Example 2

When the code below is run:

```
==========================================================================
d2 = dish()
d2.name = 'Bamboo Shoot salad'
d2.description = '(ไทย) ซุปหน่อไม้'

print(d2.name)
print(d2.description)
==========================================================================
```

it results

```
==========================================================================
Bamboo Shoot salad
(ไทย) ซุปหน่อไม้
==========================================================================
```

Note if Thai alphabets seem to be an issue, check out encoding, e.g., 'utf8'.

P2. Method. Further develop class `dish` (from P1). Add a method `info`, which composes name and description to a string as

'name: *name*; description: *description*'

and returns this string. *The underlined italic font* indicates a corresponding attribute value, not an exact text.

Example

When the code below is run:

```
=========================================================================
d1 = dish()
d1.name = 'Chicken galangal soup'
d1.description = '(thai) tom kha kai'
s = d1.info()


print(type(d1))
print(type(s))
print(s)
=========================================================================
```

it results

```
=========================================================================
<class '__main__.dish'>
<class 'str'>
name: Chicken galangal soup; description: (thai) tom kha kai
=========================================================================
```

P3. Method. Further develop class `dish` (from P2). Add an attribute price (as a floating-point number with a default value `100`). Also, add a method `discount`. The discount method takes a discount rate and returns the price after the discount.

Example

When the code below is run:

```
=========================================================================
d1 = dish()
d1.name = 'Chicken galangal soup'
d1.description = '(thai) tom kha kai'
s = d1.info()
```

```
print(type(d1))
print(type(s))
print(s)
print(d1.price)
p = d1.discount(25)
print(p)
```

==========

it results

==========

```
<class '__main__.dish'>
<class 'str'>
name: Chicken galangal soup; description: (thai) tom kha kai
100
75.0
```

==========

P4. Constructor. Further develop class `dish` (from P3). Modify its construction `__init__` to allow 3 arguments: `nam`, `desc`, and `pric` with default values `'signature dish'`, `'healthy and tasty'`, and `100`, respectively. These 3 arguments are to initialize all 3 attributes: `name`, `description`, and `price`, in order.

Example

When the code below is run:

==========

```
d1 = dish()
s = d1.info()
print(s)
print(d1.price)
```

```
d2 = dish('mussel pineapple curry')
s2 = d2.info()
print(s2)
print(d2.price)

d3 = dish('basil spicy hotpot', 'jeolhon', 200)
s3 = d3.info()
print(s3)
print(d3.price)
```

==========

it results

==========

```
name: ; description:
100
name: mussel pineapple curry; description:
100
name: basil spicy hotpot; description: jeolhon
200
```

==========

P5. "Private" attribute. Further develop class `dish` (from P4). Change the attribute `price` to be "private" to limit its access. Also, add methods `get_price` to return value of the price attribute and `set_price` to update the price attribute value.

Example

When the code below is run:

==========

```
d1 = dish('Chicken galangal soup', 'tom kha kai', 150)
```

```
d1.set_price(120)
print(d1.info(), d1.get_price())
print(hasattr(d1, 'name'))
print(hasattr(d1, 'price'))
print(hasattr(d1, '_price'))
print(hasattr(d1, '__price'))
```

==========

it results

==========

```
name: Chicken galangal soup; description: tom kha kai 120
True
False
False
False
```

==========

P6. Inheritance. Based on the class `dish` (from P5), develop a class `SDish`, which inherits `dish` and has 3 prices for small, medium, and large servings. All 3 prices are private and have `set_priceS`, `get_priceS`, `set_priceM`, `get_priceM`, `set_priceL`, and `get_priceL` for their access.

Note: for smooth autograding, at the P6 header write:

```
from P5_sys import dish
```

and make a copy of `P5.py` and rename it `P5_sys.py`.

Example

When the code below is run:

==========

```
print(issubclass(SDish, dish))
```

```
d1 = SDish('Chicken galangal soup', 'tom kha kai')
print(d1.info())
d1.set_priceS(80)
d1.set_priceM(100)
d1.set_priceL(120)
print(d1.get_priceS(), d1.get_priceM(), d1.get_priceL())

d2 = SDish('Bamboo shoot salad', 'soop noh mai')
print(d2.info())
d2.set_priceS(40)
d2.set_priceM(60)
d2.set_priceL(80)
print(d2.get_priceS(), d2.get_priceM(), d2.get_priceL())

print(d1.info())
print(d1.get_priceS(), d1.get_priceM(), d1.get_priceL())
```

==========

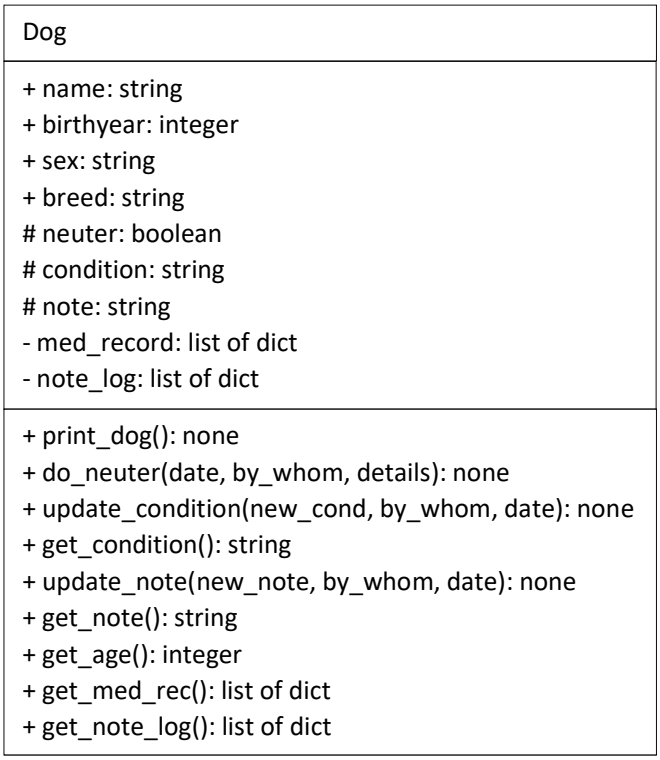
it results

```
==========
True
name: Chicken galangal soup; description: tom kha kai
80 100 120
name: Bamboo shoot salad; description: soop noh mai
40 60 80
name: Chicken galangal soup; description: tom kha kai
80 100 120
==========
```

P7. Imagine we are helping a dog shelter to develop a program to keep track of stray dogs coming to the shelter. Each dog needs to be profiled for: `name`, `birthyear` (or estimation, it is for estimating his/her age), `sex`, `breed` (or guess, it is for medical concern as well as adoption), `spay_neuter` (for

arranging the operation), `condition` (for special medical/psychological attention), `note` (miscellaneous note for anything, including personality, aggression, adoption arrangement, etc.).

The `note` and `condition` has to be kept on records for every update. The design of the class `Dog` is as shown in the class diagram below (`+ public`, `# protected`, `- private`).



Since `neuter`, `condition`, and `note` should be kept on record, to ensure data consistency it is better to make them protected and accessible through designated methods. The records of condition and note should be mostly internally maintained, both records `med_rec` and `note_log` are designed to be private. Both `med_rec` and `note_log` are lists of dictionaries:

```
med_rec = [{"condition": <condition text>,  "recorded by": <a person name>, "date": <date in "YY/MM/DD" format>, ...}]
```

`note_log = [{"note": <note text >, "recorded by": <a person name>, "date": <date in "YY/MM/DD" format>, ...}]`.

A *red italic comics font* indicates attribute value, while a `blue consolas font` indicates an exact string.

The methods of class `Dog` are:

* method `print_dog` to print all basic dog information: all public and protected attributes. It prints out in the format:

<name> <birthyear> <sex> <breed> `; neuter=` *<neuter>* `; condition=` *<condition>* `; note=` *<note>*

* method `do_neuter` to update neuter information (set `neuter` to `True`) and record this to `med_record` with date, veterinarian in charge, and details (if any).

* method `update_condition` to update a dog medical condition and record this to `med_record` with a new condition to update, a person in charge, and the date.

* method `get_condition` to get a dog medical condition as a string.

* method `update_note` to update a dog miscellaneous note and record this to `note_log` with a new condition to update, a person in charge, and the date.

* method `get_note` to get a dog miscellaneous note as a string.

* method `get_age` to calculate a dog age (from `birthyear` and a current year) and return the age as an integer. Estimate the age by

current year – birthyear.

Hint: module time has `time.strftime("%Y")`*, which returns the current year.*

* method `get_med_rec` to get an entire list of `med_rec`. !Prevent the private list from unintended change. See Example, Test 3.

* method `get_note_log` to get an entire list of `note_log`. !Prevent the private list from unintended change. See Example, Test 7.

Example

When the code below is run:

```
==========================================================================
d = Dog()


d.name = "Lucky"
d.birthyear = 2008            # birthdate or the estimation
d.sex = "F"                   # dog sex M (male)/ F (female)
d.breed = "golden retriever"  # dog breed


print("1. Test print_dog")
d.print_dog()


print("2. Test do_neuter")
print(d.get_med_rec())
d.do_neuter("2009/4/15", "Vet One")
d.print_dog()
print(d.get_med_rec())


print("3. Test if med_rec is safe")
a = d.get_med_rec()
a.append("test")
print(a)
print(d.get_med_rec())


print("4. Test update condition")
d.update_condition("Deceased", "Vet Home", "2019/9/1")
d.print_dog()
print(d.get_med_rec())
```

```
print("5. Test get condition")
print(d.get_condition())

print("6. Test update note")
d.update_note("Nice, friendly, love to play", "Trainer", "2010/8/16")
d.print_dog()
print(d.get_note())
print(d.get_note_log())

print("7. Test if note_log is safe")
a = d.get_note_log()
a.append("test")
print(a)
print(d.get_note_log())

print("8. Test multiple updates")
d.update_condition("Ready to be reborn", "The Wheel", "2019/10/19")
d.update_note("Loved and missed", "Trainer 2", "2019/10/19")

d.print_dog()
print(d.get_med_rec())
print(d.get_note_log())

print("9. Test get age")
print(d.get_age())
```

==========

it results

==========

```
1. Test print_dog
Lucky 2008 F golden retriever ; neuter= False ; condition=  ; note=
2. Test do_neuter
[]
```

```
Lucky 2008 F golden retriever ; neuter= True ; condition=  ; note=

[{'condition': 'Get neutered', 'date': '2009/4/15', 'recorded by': 'Vet One'}]

3. Test if med_rec is safe

[{'condition': 'Get neutered', 'date': '2009/4/15', 'recorded by': 'Vet One'}, 'test']

[{'condition': 'Get neutered', 'date': '2009/4/15', 'recorded by': 'Vet One'}]

4. Test update condition

Lucky 2008 F golden retriever ; neuter= True ; condition= Deceased ; note=

[{'condition': 'Get neutered', 'date': '2009/4/15', 'recorded by': 'Vet One'}, {'condition': 'Deceased', 'date': '2019/9/1', 'recorded by': 'Vet Home'}]

5. Test get condition

Deceased

6. Test update note

Lucky 2008 F golden retriever ; neuter= True ; condition= Deceased ; note= Nice, friendly, love to play

Nice, friendly, love to play

[{'note': 'Nice, friendly, love to play', 'date': '2010/8/16', 'recorded by': 'Trainer'}]

7. Test if note_log is safe

[{'note': 'Nice, friendly, love to play', 'date': '2010/8/16', 'recorded by': 'Trainer'}, 'test']

[{'note': 'Nice, friendly, love to play', 'date': '2010/8/16', 'recorded by': 'Trainer'}]

8. Test multiple updates

Lucky 2008 F golden retriever ; neuter= True ; condition= Ready to be reborn ; note= Loved and missed

[{'condition': 'Get neutered', 'date': '2009/4/15', 'recorded by': 'Vet One'}, {'condition': 'Deceased', 'date': '2019/9/1', 'recorded by': 'Vet Home'}, {'condition': 'Ready to be reborn', 'date': '2019/10/19', 'recorded by': 'The Wheel'}]
```

```
[{'note': 'Nice, friendly, love to play', 'date': '2010/8/16', 'recorded by': 'Trainer'}, {'note': 'Loved and missed', 'date': '2019/10/19', 'recorded by': 'Trainer 2'}]

9. Test get age

11
```

==========

Notice that `neuter`, `condition`, and `note` are initialized to `False`, `""`, and `""`, respectively. Both `med_rec` and `note_log` are initialized to empty lists.

==========

EXTRA PROBLEMS

(No Grader, as of Oct 19th, 2019. I am exhausted.)

==========

P8. To make class `Dog` more persistent, add two methods: `save` and the constructor `__init__`.

* method `save` to save all the information into a specified file.

* constructor `__init__` is to initialize all attributes. Let make it a bit flexible:

(a) if its argument is a string, then the record is just a retrieved record of a sheltered dog. The string argument is a file name, used a content in the file to initialize all attributes.

(b) if its argument is a dictionary, then the record is new for a new coming dog. The argument contains key-value pairs whose keys correspond to public attributes and whose values to initialize them. Assuming the initial dict has all public attributes. Protected and private attributes (in case of dictionary argument) are initialized as follows:

`neuter` is initialized to `False`,

`condition` is initialized to `"Arranging a med exam"`,

`note` is initialized to `"New coming"`,

`med_rec` is initialized to `[{"condition": <the current condition text, see condition above>, "recorded by": "Automatic", "date": <current date in "YY/MM/DD" format>}]`,

`note_log` is initialized to `[{"note": <the current note text, see note above>, "recorded by": "Automatic", "date": <current date in "YY/MM/DD" format>}]` .

The final class diagram of class Dog now becomes: (**bold font** indicates the addition.)

Dog
+ name: string + birthyear: integer + sex: string + breed: string # neuter: boolean # condition: string # note: string - med_record: list of dict - note_log: list of dict
+ print_dog(): none + do_neuter(date, by_whom, details): none + update_condition(new_cond, by_whom, date): none + get_condition(): string + update_note(new_note, by_whom, date): none + get_note(): string + get_age(): integer + get_med_rec(): list of dict + get_note_log(): list of dict **+ save(fname): none** **+ __init__(init_params): none**

Note: the specific format of a saved content is left to you. You can design and custom-make it, but off-the-shelf modules, e.g., `dbm` and `pickly`, could be handy.

P9. According to P8, it is good that we have class `Dog` ready for dog records, but this is far from being done. A shelter needs to keep track of all of its sheltered dogs, their cages, date admitted to the shelter, as well as keep track of dog-cage assignment records (a dog can be moved to other cages) and the adoption record. (Yeah! some sheltered dogs can find homes.) We will leave donation, staff and volunteer, other management matters out of this (for now).