Classes in Python

Basic Syntax

Based on `examples/Python/fall_asleep_1.py`

```
class Sheep:
    x = 0

    def flock(self):
        self.x = self.x + 1
        print("So far", self.x)

dolly = Sheep()

dolly.flock()
dolly.flock()
dolly.flock()
```

```
$ python3 fall_asleep.py
So far 1
So far 2
So far 3
```

- `class` is a reserved word
- This is the template for making `Sheep` objects
- Each `Sheep` object has a bit of data
- Each `Sheep` object has a bit of code
- Construct a `Sheep` object and store in `dolly`
- Tell the object to run the `flock()` code within it (three times)
- `dolly.flock()` ⟺ `Sheep.flock(dolly)`

Basic Syntax

Based on `examples/Python/fall_asleep_1.py`

```
print(type(dolly))
print(dir(dolly))

print(type(Sheep))
print(dir(Sheep))
```

- Elements enclosed in double underscores are Python internal variables you can mostly ignore

```
<class '__main__.Sheep'>
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'flock', 'x']
<class 'type'>
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'flock', 'x']
```

`__init__` – Constructor

Based on `examples/Python/fall_asleep_1.py`

```
print(help(dolly.__init__))
```

```
Help on method-wrapper:

__init__ = <method-wrapper '__init__' of Sheep object>
    Initialize self.  See help(type(self)) for accurate signature.
```

```
class Sheep:
    x = 0

    def __init__(self):
        """I construct Sheep instances"""
        print('I am constructed')

    ...

print(help(dolly.__init__))
```

`__init__()` is a reserved method: it is the constructor and is always called when an object is created

```
Help on method __init__ in module __main__:

__init__() method of __main__.Sheep instance
    I construct Sheep instances
```

`__del__` – Destructor

Based on `examples/Python/fall_asleep_1.py`

```
class Sheep:

    ...

    def __del__(self):
        print('I am destructed', self.x)

    ...

dolly = Sheep()

dolly.flock()
dolly.flock()
dolly.flock()

dolly = 13

print('dolly contains', dolly)
```

`__del__()` is a reserved method: it is the destructor

```
I am constructed
So far 1
So far 2
So far 3
I am destructed 3
dolly contains 13
```

Constructor and Destructor

Based on `examples/Python/fall_asleep_2.py`

```
class Sheep:
    x = 0
    name = "Just another sheep"

    def __init__(self, name=name):
        """I construct Sheep instances"""
        self.name = name
        print('{0} speaking: I am constructed :-)'.format(self.name))

    def flock(self):
        self.x = self.x + 1
        print('{0} flock count: {1}'.format(self.name, self.x))

    def __del__(self):
        print('{0} speaking: I am destructed :-('.format(self.name))

dolly = Sheep()
dolly.flock()
dolly = 13

dolly = Sheep("Dolly")
dolly.flock()
montauciel = Sheep("Montauciel")
montauciel.flock()
montauciel.flock()
print(montauciel.x)
dolly = 'a'
```

```
$ python3 fall_asleep_2.py
Just another sheep speaking: I am constructed :-)
Just another sheep flock count: 1
Just another sheep speaking: I am destructed :-(
Dolly speaking: I am constructed :-)
Dolly flock count: 1
Montauciel speaking: I am constructed :-)
Montauciel flock count: 1
Montauciel flock count: 2
2
Dolly speaking: I am destructed :-(
Montauciel speaking: I am destructed :-(
```

`__str__`

Based on `examples/Python/fall_asleep_2.py`

```
class Sheep:
    ...
    def __str__(self):
        info = '(x = {0}, name = {1})'.format(str(self.x), str(self.name))
        return info
    ...

montauciel = Sheep("Montauciel")
montauciel.flock()
print(montauciel)
```

```
$ python3 fall_asleep_2.py
Montauciel speaking: I am constructed :-)
Montauciel flock count: 1
(x = 1, name = Montauciel)
Montauciel speaking: I am destructed :-(
```

- Returns a string representation of the object
- If a class provides a method named `__str__`, it overrides the default behaviour of the built-in `str` function
- Printing the object implicitly invokes `__str__` on it, so defining `__str__` also changes the behaviour of `print`
- When writing a new class, start by implementing `__init__` and then `__str__`: it is useful for debugging purposes

Inheritance

Based on `examples/Python/fall_asleep_2.py`

```
class Sheep:
    ...

class Lamb(Sheep):
    siblings = 0

    def set_siblings(self, s):
        self.siblings = s
        print('{0} speaking: I have {1} siblings'.format(self.name, self.siblings))
        for i in range(s):
            self.flock()

montauciel = Sheep("Montauciel")
montauciel.flock()
dolly = Lamb("Dolly")
dolly.flock()
dolly.set_siblings(3)
print(montauciel)
```

Inheritance

Based on `examples/Python/fall_asleep_2.py`

```
class Sheep:
    ...

class Lamb(Sheep):
    siblings = 0

    def set_siblings(self, s):
        self.siblings = s
        print('{0} speaking: I have {1} siblings'.format(sel
        for i in range(s):
            self.flock()

montauciel = Sheep("Montauciel")
montauciel.flock()
dolly = Lamb("Dolly")
dolly.flock()
dolly.set_siblings(3)
print(montauciel)
```

```
Montauciel speaking: I am constructed :-)
Montauciel flock count: 1
Dolly speaking: I am constructed :-)
Dolly flock count: 1
Dolly speaking: I have 3 siblings
Dolly flock count: 2
Dolly flock count: 3
Dolly flock count: 4
(x = 1, name = Montauciel)
Montauciel speaking: I am destructed :-(
Dolly speaking: I am destructed :-(
```

Operator Overloading/Overriding

Python allows operator overloading/overriding

- E.g., provide a method named `__add__` to override the addition operator +

```
class Vector:
    # All the rest of a 2D vector class...

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
```

```
>>> v1 = Vector(1, 2)
>>> v2 = Vector(-1, 5)
>>> print(v1 + v2)
(0, 7)
>>> print(v1.__add__(v2))
(0, 7)
```

- `__sub__` can be provided to override the subtraction operator
- To override the behaviour of the multiplication operator, define a method named `__mul__`, or `__rmul__`, or both

Operator Overloading/Overriding

```
class Vector:
    # All the rest of a 2D vector class...

    def __mul__(self, other):
        return self.x * other.x + self.y * other.y
```

```
>>> v1 = Vector(1, 2)
>>> v2 = Vector(-1, 5)
>>> print(v1 * v2)
9
>>> print(2.*v1)
TypeError: unsupported operand type(s)
for *: 'float' and 'Vector'
```

```
class Vector:
    # All the rest of a 2D vector class...

    def __mul__(self, other):
        return self.x * other.x + self.y * other.y

    def __rmul__(self, other):
        return Vector(other * self.x, other * self.y)
```

```
>>> v1 = Vector(1, 2)
>>> v2 = Vector(-1, 5)
>>> print(v1 * v2)
9
>>> print(2.*v1)
(2.0, 4.0)
>>> print(2*v1)
(2, 4)
>>> print(v1*2.)
AttributeError: 'float' object has no
attribute 'x'
```

Operator Overloading/Overriding

- The built-in function `isinstance()` function returns `True` if the specified object is of the specified type, otherwise `False`
- `self.__class__` is automatically available
- The raise keyword is used to raise an exception

```
>>> v1 = Vector(1, 2)
>>> v2 = Vector(-1, 5)
>>> print(v1 * v2)
9
>>> print(2.*v1)
(2.0, 4.0)
>>> print(2*v1)
(2, 4)
>>> print(v1*2.)
(2.0, 4.0)
>>> print(v1*2)
(2, 4)
```

```
class Vector:
    # All the rest of a 2D vector class...

    def __mul__(self, other):
        if isinstance(other, self.__class__):
            return other.x * self.x + other.y * self.y
        elif isinstance(other, int):
            return Vector(self.x * other, self.y * other)
        elif isinstance(other, float):
            return Vector(self.x * other, self.y * other)
        else:
            raise TypeError("Unsupported operand type(s) for *: '{}' and '{}'").format(self.__class__, type(other))

    def __rmul__(self, other):
         return Vector(other * self.x, other * self.y)
```

(Runtime) Polymorphism

Based on `examples/Python/polymorphism.py`

```
class Vector:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __mul__(self, other):
        if isinstance(other, self.__class__):
            return other.x * self.x + other.y * self.y
        elif isinstance(other, int):
            return Vector(self.x * other, self.y * other)
        elif isinstance(other, float):
            return Vector(self.x * other, self.y * other)
        else:
            raise TypeError("Unsupported operand type(s) for *: '{}' and '{}'").format(self.__class__, type(other))

    def __rmul__(self, other):
         return Vector(other * self.x, other * self.y)

def line (m, x, q):
    return m * x + q
```

```
>>> v1 = Vector(1, 2)
>>> v2 = Vector(-1, 5)
>>> print(line(3, 2, 1))
7
>>> print(line(2, v1, v2))
(1, 9)
>>> print(line(v1, v2, 1))
10
>>> print(line(v1, 2, v2))
(1, 9)
```

Other Python Operators

Operator	Expression	Internally
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder	`p1 % p2`	`p1.__mod__(p2)`
Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`
Bitwise OR	`p1 | p2`	`p1.__or__(p2)`
Bitwise XOR	`p1 ^ p2`	`p1.__xor__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

Operator	Expression	Internally
Less than	`p1 < p2`	`p1.__lt__(p2)`
Less than or equal to	`p1 <= p2`	`p1.__le__(p2)`
Equal to	`p1 == p2`	`p1.__eq__(p2)`
Not equal to	`p1 != p2`	`p1.__ne__(p2)`
Greater than	`p1 > p2`	`p1.__gt__(p2)`
Greater than or equal to	`p1 >= p2`	`p1.__ge__(p2)`

Class Variables: Mutables

Based on `examples/Python/fall_asleep_2.py`

```
class Sheep:
    x = 0

    ...

    def flock(self):
        self.x = self.x + 1
        print('{0} flock count: {1}'.format(self.name, self.x))

class Lamb(Sheep):
    ...

    def set_siblings(self, s):
        ...
        for i in range(s):
            self.flock()


montauciel = Sheep("Montauciel")
montauciel.flock()
dolly = Lamb("Dolly")
dolly.flock()
dolly.set_siblings(3)
print(montauciel)
```

```
Montauciel speaking: I am constructed :-)
Montauciel flock count: 1
Dolly speaking: I am constructed :-)
Dolly flock count: 1
Dolly speaking: I have 3 siblings
Dolly flock count: 2
Dolly flock count: 3
Dolly flock count: 4
(x = 1, name = Montauciel)
Montauciel speaking: I am destructed :-(
Dolly speaking: I am destructed :-(
```

Notice that `flock()` increases `x` but each `Sheep` instance has its own `x`

- We do not count members of a single flock but start a flock with each `Sheep` instance

Class Variables: Mutables

- With **mutable objects**, such as lists and dictionaries, the behaviour is different
  - Remember: **mutable** objects can change their value but keep their `id()`
  - They behave as “shared data”

Class Variables: Mutables

Based on `examples/Python/fall_asleep_3.py`

```
class Sheep:
    members = []

    ...
    def flock(self):
        self.members.append(self.name)
        print('{0} flock count: {1}'.format(self.name, len(self.members)))

    def __str__(self):
        info = '(members = {0}, name = {1})'.format(str(self.members), str(self.name))
        return info

class Lamb(Sheep):
    siblings = 0

    def __str__(self):
        info = '(members = {0}, name = {1}, siblings = {2})'.format(str(self.members), str(self.name), str(self.siblings))
        return info

    ...

montauciel = Sheep("Montauciel")
montauciel.flock()
dolly = Lamb("Dolly")
dolly.flock()
dolly.set_siblings(3)
print(dolly)
print(montauciel)
```

Class Variables: Mutables

Based on `examples/Python/fall_asleep_3.py`

```
class Sheep:
    members = []

    ...
    def flock(self):
        self.members.append
        print('{0} flock co

    def __str__(self):
        info = '(members =
        return info

class Lamb(Sheep):
    siblings = 0

    def __str__(self):
        info = '(members =
        return info

    ...

montauciel = Sheep("Montauc
montauciel.flock()
dolly = Lamb("Dolly")
dolly.flock()
dolly.set_siblings(3)
print(dolly)
print(montauciel)
```

```
$ python3 fall_asleep_3.py
Montauciel speaking: I am constructed :-)
Montauciel flock count: 1
Dolly speaking: I am constructed :-)
Dolly flock count: 2
Dolly speaking: I have 3 siblings
Dolly flock count: 3
Dolly flock count: 4
Dolly flock count: 5
(members = ['Montauciel', 'Dolly', 'Dolly', 'Dolly', 'Dolly'], name = Dolly, siblings = 3)
(members = ['Montauciel', 'Dolly', 'Dolly', 'Dolly', 'Dolly'], name = Montauciel)
Montauciel speaking: I am destructed :-(
Dolly speaking: I am destructed :-(
```

Class Variables: Mutables

- With **mutable objects**, such as lists and dictionaries, the behaviour is different
  - Remember: **mutable** objects can change their value but keep their `id()`
  - They behave as “shared data”
  - Place instance-specific mutable variables in the `__init__`

Class Variables: Mutables

Based on `examples/Python/fall_asleep_3.py`

```
class Sheep:
    members = []

    def __init__(self, name=name):
        """I construct Sheep instances"""
        self.name = name
        self.owners = []
        print('{0} speaking: I am constructed :-)'.format(self.name))

    def add_owner(self, shepard):
        self.owners.append(shepard)
        print('{0} owners count: {1}'.format(self.name, len(self.owners)))

    ...

class Lamb(Sheep):
    ...

montauciel = Sheep("Montauciel")
montauciel.flock()
montauciel.add_owner("Montgolfier Bros.")
dolly = Lamb("Dolly")
dolly.flock()
dolly.add_owner("Mary")
print(dolly.owners)
print(montauciel.owners)
```

```
Montauciel speaking: I am constructed :-)
Montauciel flock count: 1
Montauciel owners count: 1
Dolly speaking: I am constructed :-)
Dolly flock count: 2
Dolly owners count: 1
['Mary']
['Montgolfier Bros.']
Montauciel speaking: I am destructed :-(
Dolly speaking: I am destructed :-(
```

More on Variables: local, `nonlocal`, `global`

- Another way to count sheep instances would be to have `flock` live/defined outside the `Sheep` class and having the `Sheep` constructor update `flock`

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
# Notice we did not define spam here, in the global scope
```

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

More on Variables: local, `nonlocal`, `global`

Based on `examples/Python/fall_asleep_4.py`

```
class Sheep:
    name = "Just another sheep"

    def __init__(self, name=name):
        """I construct Sheep instances"""
        self.name = name
        self.owners = []
        self.flock()
        print('{0} speaking: I am constructed :-)'.format(self.name))

    def flock(self):
        global flock
        flock += 1
        print('{0} flock count: {1}'.format(self.name, flock))
    ...

class Lamb(Sheep):
    ...

flock = 0
montauciel = Sheep("Montauciel")
dolly = Lamb("Dolly")
dolly.set_siblings(3)
print(flock)
flock += 1
print(flock)
```

```
Montauciel flock count: 1
Montauciel speaking: I am constructed :-)
Dolly flock count: 2
Dolly speaking: I am constructed :-)
Dolly speaking: I have 3 siblings
Dolly flock count: 3
Dolly flock count: 4
Dolly flock count: 5
5
6
Montauciel speaking: I am destructed :-(
Dolly speaking: I am destructed :-(
```

It works, but `flock` is not private and not just `Sheep` instances can modify it

Does Python even have private variables?
NO!

More on Variables: _ Prefix and __ Prefix

- To emulate private variables, use the __ prefix (with one trailing _ at most)
  - Python **mangles** the names of variables like `__foo` (which becomes `_classname__foo`) so they are not easily visible to code outside the class containing them
- By the same **convention**, the _ prefix means "stay away even if you are not technically prevented from doing so"

```
class Sheep:
    def __init__(self, name):
        self.__name = name

    def displayName(self):
        print(self.__name)

dolly = Sheep("Dolly")
dolly.displayName()

# Raises error
print(dolly.__name)

# Would not raise and error
print(dolly._Sheep__name)
```

Bottom line: do not play around with variables from another class that look like `__foo` or `_bar`

Empty Class and Deleting Attributes (`del`)

- For a data type similar to the C “struct,” use an empty class definition
- Attributes (not necessarily of a class, but of objects in general) may be deleted

```
class Sheep:
    pass

# Create an empty record
dolly = Sheep()

# Add and fill in its fields
dolly.name = 'Dolly'
dolly.owners = ['Mary']
dolly.siblings = 3
print(dolly.name, dolly.owners, dolly.siblings)

# Remove a field
del dolly.siblings
print(dolly.name, dolly.owners, dolly.siblings)
```

```
Dolly ['Mary'] 3
Traceback (most recent call last):
  File "[...]del.py", line 15, in <module>
    print(dolly.name, dolly.owners, dolly.siblings)
AttributeError: 'Sheep' object has no attribute 'siblings'
```

Static Methods

Based on `examples/Python/fall_asleep_4.py`

- Static methods are methods that belong to a class but do not have access to `self` and hence do not require an instance to work
- Denote these with the line `@staticmethod` before defining them

```
class Sheep:
    ...

    @staticmethod
    def try_to_talk():
        print("Baaaahhh!")

# Class Sheep can talk
Sheep.try_to_talk()

# As can Sheep instances
dolly = Sheep()
dolly.try_to_talk()
```

```
Baaaahhh!
Just another sheep speaking: I am constructed :-)
Baaaahhh!
```

More on Inheritance

- Multiple inheritance is supported: `class Child(Parent1, Parent2):`
  - Search for attributes inherited from a parent class: depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy
- Two built-in functions that work with inheritance:
  1. `isinstance()` checks an instance's type
     - `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`
  2. `issubclass()` checks class inheritance
     - `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`
     - `issubclass(float, int)` is `False` since `float` is not

SciPy

Based on `examples/Python/6-SciPy.ipynb`