Python Lesson

In [1]:

```
%%javascript
$.getScript("https://kmahelona.github.io/ipython_notebook_goodies/ipython_notebook_t
```

```
<IPython.core.display.Javascript object>
```

In [2]:

```
import os
assert(os.sys.version[0]=='3')
```

Class

Different Programming Paradigms

3 Different approaches, we will focus on OOP, and will use examples to illustrate the difference

- Procedure Programming
- Object Oriented Programming
- Functional Programming

Procedural Programming

- focus on data
- input and output
- use functions/subroutines to process input and produce output
- good software system often decouple into separate modules to complete different functionalities

Object Oriented Programming

- model the world with class and objects
- abstract the world into an hierarchy of classes
- use class as template, create objects (class instance)
- objects interact with each other to achieve some objectives

OOP Basic Concepts

- abstraction
  - find similar objects, focus the important aspect and leave out the details
  - important aspects are a. what is it? b. what can it do?
  - attribute (data member)
  - methods (member functions, etc)
  - Examples:
  - A cat is an animal has whiskers, paws, tails and it can climb a tree and mew
  - A dog has paws, tails but no whiskers! it can bark.
  - Implementation in Python:
  - cat.color = "yellow"
  - cat.climb()
  - dog.bark()
  - through abstraction, you derive different classes: cat, dog, ...
  - through instantiation, you get objects / instances: cat 1, cat 2 ...
- encapsulation:
  - why do we encapsulate implemention?
  - privacy and delegation
  - car (class) can move from one place to another
  - we let car do its job:
  - car.add_fuel(), car.move()
  - add_fuel method will either add gas or charge itself (electrical cars)
  - encapsulation help a smooth migration from one technology to another
- inheritance:
  - model an is-a relationship.
  - create a hierarchy of classes
  - dog is subclass of animal, golden retriever is a subclass of dog
- polymorphism:
  - ploymorphism means many forms.
  - a base class's function can take many forms
  - example:
  - car.add_fuel()

Functional Programming

- focus on data transformation in parellel
- create anonymous functions on the fly
- map and reduce
- a lot more...

In [2]:

```
# a deck of cards  (?)

# 4 groups

# hearts ?
# spade ?
# diamond ?
# clubs ?

# count total each category
# heart_count += 1
# club_count += 10
# heart_count += 7
# ....

# ...


# 1 person  catergorize 4 groups
# 4 people.
# p1: heart: 1+3+6...
# p2: club: 2+4+5...
```

Example

In [3]:

```
# procedure programming
def f2c(fahrenheit):
    celsius = ( fahrenheit - 32. ) * 5 / 9
    return celsius

def c2f(celsius):
    fahrenheit = 9.0 / 5.0 * celsius + 32.
    return fahrenheit
```

In [4]:

```
f2c(95)
```

Out[4]:

```
35.0
```

In [5]:

```
c2f(35)
```

Out[5]:

```
95.0
```

In [6]:

```
print("convert from celsius to fahrenheit")
for temperature in range(0, 40, 5):
    print(temperature, c2f(temperature))
```

```
convert from celsius to fahrenheit
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
```

In [7]:

```
# Object Oriented Programming
class Converter():  # pay attention to syntax !!! different than function
    count = 0 # class variable
    def __init__(self, name=""):
        print("__init__ called with {}".format(name))
        self.result = 0  # attribute
        self.name = name
        Converter.count += 1

    def get_name(self):
        return self.name

    def to_celsius(self, fahrenheit):
        """ convert fahrenheit to celsius """
        self.result = ( fahrenheit - 32. ) * 5 / 9
        print(self.result)
        return self

    def to_fahrenheit(self, celsius):
        self.result = 9.0 / 5.0 * celsius + 32.
        print(self.result)
        return self

    def last_result(self):
        print(self.result)
```

In [8]:

```
# class_name() : constructor  create an object from class
c = Converter(name="converter #1")  # an converter object is called. and __init__ i
c.to_celsius(95)
c.to_fahrenheit(35)
c.last_result()
print(c.name)
d = Converter(name="converter #2")
print(d.name)
# class variable
c.count, d.count
```

```
__init__ called with converter #1
35.0
95.0
95.0
converter #1
__init__ called with converter #2
converter #2
```

Out[8]:

```
(2, 2)
```

In [10]:

```
# Functional Programming -- it's totally ok if you feel dizzy, just ignore this par
# map input celius to fahrenheit
# lambda <input> : <transformation as output>
# map input from a sequence of cecius to a sequence of fahranheit
# map(<transformation>, <a sequence>)
list(map(lambda a: 9.0 / 5.0 * a + 32., range(0, 40, 5)))
```

Out[10]:

```
[32.0, 41.0, 50.0, 59.0, 68.0, 77.0, 86.0, 95.0]
```

In [10]:

```
# what if I just want to see the total output?
# map and reduce

# reduce: sum the sequence
# reduce(function, sequence) -> value
# you can directly embed anonymous function using lambda syntax
from functools import reduce
reduce(lambda x,y:x+y, map(lambda c: 9.0 / 5.0 * c + 32., range(0, 40, 5)) )
```

Out[10]:

```
508.0
```

More examples on Class

In [14]:

```python
class Car():
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def add_fuel(self):
        # this can be overriden by subclass
        print("Car is adding fuel")

class Jeep(Car):
    def __init__(self, color):
        super().__init__("Jeep", color)
#          Car.__init__(self, "Jeep", color)

    def add_fuel(self):
        print("drinking gas...")

class Tesla(Car):
    def __init__(self, color):
        super().__init__("Tesla", color)

    def add_fuel(self):
        print("charging...")
```

In [16]:

```python
f = Jeep("Green")
print(f.color, f.name)
f.add_fuel()

t = Tesla("Black")
print(t.color, t.name)
t.add_fuel()

c=Car("","")
c.add_fuel()
```

```
Green Jeep
drinking gas...
Black Tesla
charging...
Car is adding fuel
```

Summary

- class definition
- class initialization
- define attributes
- define methods
- class inheritance
- superclass's method can be overriden by subclass's same method
- through inheritance, subclass exhibit polymorphism
- benefit:
    - division of work
    - organize information and access to it
    - smooth feature migration

In [ ]:

In [ ]:

Homework

use procedure programming, write a function to find all complete numbers in the range(0,100)

- a complete number is a number equals to the sum of its factors: 6 = 1 + 2 + 3
- to test if a is a factor of b: if b % a == 0

use OOP, write a class to encode a string to a number according to this

- input 1: "aBc"
    - transform "aBc" to all upper cases "ABC"
    - transform "ABC" -> 1,2,3
    - transform 1, 2, 3 to 6 by sum all the numbers 1+2+3=6
    - output 6
- input 2: "attitude" , output: 100

bonus points: use functional programming to implement the above encoding

In [18]:

```
# www.python.org  tutorial
```

In [ ]:

```

```

Python Lesson

In [1]:

```

```

```
<IPython.core.display.Javascript object>
```

In [2]:

```

```

Class

Different Programming Paradigms

3 Different approaches, we will focus on OOP, and will use examples to illustrate the difference

- Procedure Programming
- Object Oriented Programming
- Functional Programming

Procedural Programming

- focus on data
- input and output
- use functions/subroutines to process input and produce output
- good software system often decouple into separate modules to complete different functionalities

Object Oriented Programming

- model the world with class and objects
- abstract the world into an hierarchy of classes
- use class as template, create objects (class instance)
- objects interact with each other to achieve some objectives

OOP Basic Concepts

- abstraction
    - find similar objects, focus the important aspect and leave out the details
    - important aspects are a. what is it? b. what can it do?
    - attribute (data member)
    - methods (member functions, etc)
    - Examples:
    - A cat is an animal has whiskers, paws, tails and it can climb a tree and mew
    - A dog has paws, tails but no whiskers! it can bark.
    - Implementation in Python:
    - cat.color = "yellow"
    - cat.climb()
    - dog.bark()
    - through abstraction, you derive different classes: cat, dog, ...
    - through instantiation, you get objects / instances: cat 1, cat 2 ...
- encapsulation:
    - why do we encapsulate implemention?
    - privacy and delegation
    - car (class) can move from one place to another
    - we let car do its job:
    - car.add_fuel(), car.move()
    - add_fuel method will either add gas or charge itself (electrical cars)
    - encapsulation help a smooth migration from one technology to another
- inheritance:
    - model an is-a relationship.
    - create a hierarchy of classes
    - dog is subclass of animal, golden retriever is a subclass of dog
- polymorphism:
    - ploymorphism means many forms.
    - a base class's function can take many forms
    - example:
    - car.add_fuel()

Functional Programming

- focus on data transformation in parellel
- create anonymous functions on the fly
- map and reduce
- a lot more...

In [2]:

Example

In [3]:

In [4]:

Out[4]:

```
35.0
```

In [5]:

Out[5]:

```
95.0
```

In [6]:

```
convert from celsius to fahrenheit
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
```

In [7]:

In [8]:

```
__init__ called with converter #1
35.0
95.0
95.0
converter #1
__init__ called with converter #2
converter #2
```

Out[8]:

```
(2, 2)
```

In [10]:

Out[10]:

```
[32.0, 41.0, 50.0, 59.0, 68.0, 77.0, 86.0, 95.0]
```

In [10]:

Out[10]:

```
508.0
```

More examples on Class

In [14]:

In [16]:

```
Green Jeep
drinking gas...
Black Tesla
charging...
Car is adding fuel
```

Summary

- class definition
- class initialization
- define attributes
- define methods
- class inheritance
- superclass's method can be overriden by subclass's same method
- through inheritance, subclass exhibit polymorphism
- benefit:
    - division of work
    - organize information and access to it
    - smooth feature migration

In [ ]:

```
```

In [ ]:

```
```

Homework

use procedure programming, write a function to find all complete numbers in the range(0,100)

- a complete number is a number equals to the sum of its factors: 6 = 1 + 2 + 3
- to test if a is a factor of b: if b % a == 0

use OOP, write a class to encode a string to a number according to this

- input 1: "aBc"
    - transform "aBc" to all upper cases "ABC"
    - transform "ABC" -> 1,2,3
    - transform 1, 2, 3 to 6 by sum all the numbers 1+2+3=6
    - output 6
- input 2: "attitude" , output: 100

bonus points: use functional programming to implement the above encoding

In [18]:

```
```

In [ ]: