Computação Distribuída
2019 / 2020

Licenciatura em Engenharia Informática

Lab. 03 – Classes e threads em Python

1. Classes

Em Programação Orientada a Objectos, uma classe é uma estrutura que permite agrupar um conjunto de atributos e de funções para manipular esses atributos (chamados de métodos). Por exemplo, a classe seguinte define uma pessoa que possui os atributos *name* e *age* e o método *greet*.

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print("Hello, my name is %s!" % self.name)
```

A grande maioria das classes necessita sempre do método construtor (*__init__*) para inicializar os atributos. Neste caso, o construtor recebe o nome e a idade da pessoa, e guarda essa informação na instância da classe (dada pela *keyword self*).

Por fim, o método *greet* apenas faz *print* do nome guardado na instância da classe (novamente usando o *self*).

A utilização da classe faz-se por intermédio da instanciação de **objectos**. Para instanciar dois objectos da classe *Person* e utilizarmos os seus métodos poderíamos fazer:

```
>>> a = Person("José", 20)
>>> b = Person("Ana", 19)

>>> a.greet()
Hello, my name is José!
>>> b.greet()
Hello, my name is Ana!

>>> print(a.age)  # We can also access the attributes of an object
20
```

Tendo em conta a definição de classes como mencionada acima, e recorrendo à documentação do Python sobre classes (https://docs.python.org/3.5/tutorial/classes.html), resolva os seguintes exercícios:

1. Crie uma classe *Rectangle* para guardar a informação das coordenadas de um rectângulo, dadas por *(x1, y1)* e *(x2, y2)*:

2. Crie o construtor da classe *Rectangle* onde recebe como argumento as coordenadas *x1*, *y1*, *x2*, *y2*. Guarde as coordenadas nas futuras instâncias da classe usando a *keyword* ***self***.

3. Crie os métodos *width()* e *height()* que retornam, respectivamente, a largura e a altura do rectângulo. Crie pelo menos dois objectos, instância de *Rectangle*, para testar que os cálculos estão correctos.

4. Implemente o método *area()* que retorna a área do rectângulo (tamanho * largura).

5. Implemente o método *circunference()* que retorna o perímetro do rectângulo (2 * tamanho + 2 * largura).

6. Faça *print* de um dos objectos rectângulo que implementou para testar a classe. Crie o método *__str__* para que o *print* dos objectos escreva as coordenadas na forma (x1, y1) (x2, y2).

2. Herança de classes

Em Programação Orientada a Objectos, a herança é uma das formas pela qual uma classe (chamada sub-classe), herda todos os atributos e métodos da classe acima (super classe), podendo reescrever todas ou algumas funcionalidades. Por exemplo, para a classe *Person* acima, imaginemos que pretendemos criar uma classe para todas as pessoas com 10 anos de idade:

```
class TenYearOldPerson(Person):

    def __init__(self, name):
        super().__init__(name, 10)

    def greet(self):
        print("I don't talk to strangers!!")
```

A indicação que a classe *TenYearOldPerson* é uma subclasse de *Person* é dada pela indicação da classe "pai", entre parênteses, a seguir à definição da classe *TenYearOldPerson*. *Assim sendo, o* construtor da class *TenYearOldPerson* (__init__) recebe apenas o argumento do nome, mas trata de chamar o construtor da classe *Person* passando o nome e a idade igual a 10. Por fim, podemos ver que modificamos o método *greet*.

Tomando como base a classe *Rectangle* implementada nos exercícios anteriores:

1. Defina a classe *Square* como sendo uma subclasse da classe *Rectangle*.

2. Implemente o construtor da classe *Square* de modo a obter apenas a coordenada (*x1*, *y1*) e o tamanho do quadrado. Repare que terá que chamar a classe *Rectangle* com os argumentos certos quando executar o *super()*.

3. Instancie dois objectos da classe *Square*, obtenha a área e faça *print* dos objectos. Verifique que os cálculos estão correctos e que o *print* das coordenadas está coerente com o tamanho do quadrado.

3. Threads

As *threads* basicamente permitem a execução em paralelo de várias rotinas, podendo essas *threads* eventualmente ser executadas em *cores* diferentes. Em Python, a partir da versão 2.4 as *threads* são implementadas usando o módulo *threading* e o seu funcionamento é bastante similar às *threads* existentes noutras linguagens como o Java e o C#.

Considere a documentação em https://www.tutorialspoint.com/python3/python_multithreading.htm e resolva os seguintes exercícios:

1. O seguinte programa permite a criação e utilização de *threads* em Python, através da definição de uma classe *MyThread*.

```
import threading

class MyThread(threading.Thread):

    def __init__(self, name):
        super().__init__()
        self.name = name

    def run(self):
        for i in range(5):
            print(self.name, " i = ", i)


# Create new threads
thread1 = MyThread("Thread-1")
thread2 = MyThread("Thread-2")

# Start the threads
thread1.start()
thread2.start()

# Wait for threads to finish
thread1.join()
thread2.join()
print("Exiting main thread")
```

   Em Python, o método *join* permite esperar que uma *thread* termine a sua execução.

   a) Execute este código várias vezes e verifique se a ordem da execução das *threads* é sempre a mesma.
   b) O que acontece se as instruções `thread1.join()` e `thread2.join()` forem retiradas do código anterior.

c) Usando a função *time.sleep(sec)* - definida no módulo *time* – altere o código de modo que a escrita dos valores de i, seja feita de forma alternada por cada uma das *threads*.

```
Thread-1 i=0
Thread-2 i=0
Thread-1 i=1
Thread-2 i=1
    ...
```

d) Altere o programa de forma a que cada *thread* escreva no écran de forma consecutiva todos os valores de i sem sofrer qualquer interrupção. Sugestão, use o objeto *Lock* do módulo *threading*, nomeadamente os seus métodos *acquire()* e *release()*.

```
Thread-1 i=0
Thread-1 i=2
Thread-1 i=3
Thread-1 i=4
Thread-2 i=0
Thread-2 i=1
    ...
```

2. Considere a classe *SumThread*.

```
import threading

class SumThread(threading.Thread):

    def __init__(self, name, a, b):
        super().__init__()
        self.name = name
        self.a = a
        self.b = b
        self.res = 0

    def run(self):
        self.res = self.a + self.b

    def show_result(self):
        print(self.name, "sum = ", self.res)


# Create new thread
thread1 = SumThread("Thread-1", 2, 5)

# Start the threads
thread1.start()

# Show results
thread1.show_result()
```

a) Verifique qual é o funcionamento da classe `SumThread`.

b) Defina uma class e de nome `SumThreadList` que é responsável por receber uma lista de números e retornar a soma dos elementos dessa lista. Após a implementação dessa classe, deverá criar uma função de nome `sum_list2(list)` que deverá receber uma

lista inicial e criar duas threads do tipo *SumThreadList* de modo a que cada uma das threads calcule a soma de uma metade da lista. Sugestão: utilize *slicing* de modo a dividir a lista em duas. Teste com a lista [1, 2, 3, 4, 5, 6]

c) Generalize a função da alínea anterior definindo a função `sum_list(list, num_threads)` de forma a poder receber o número de *threads* que vão realizar essa soma como parâmetro.

(fim de enunciado)