Um pouco de Programação Orientada a Objetos

- O estilo de programação adotado até agora tem sido essencialmente procedimental ou imperativa
  - *mas ainda que isso possa ser uma prática comum em implementações que pretendamos simples e rápidas, em problemas de maior complexidade fará mais sentido usar técnicas de orientação por objetos – o paradigma ideal para lidar com a complexidade*
- Como já se disse antes, o Python suporta o paradigma POO, tratando tudo como objetos
  - *por exemplo, mesmos os inteiros são vistos como objetos, instâncias da classe int – para mais detalhes, consultar help(int)*

Em Python tudo são Objetos, mesmo…

É por isso, por exemplo, que tudo em Python pode ser atribuído a uma variável (guardando esta apenas a referência para o respetivo objeto)

- Objetos classe
  - *permitem-nos aceder a todos os seus membros*

    Exemplos: MinhaClasse.__doc__, MinhaClasse.x, sendo x um atributo (objeto instância) ou um método (objeto função)
  - *e, naturalmente, permitem instanciar "objetos instância"*

    Exemplo: obj=MinhaClasse()
  - *podendo ser atribuída a uma variável, pode, por exemplo, ser passada para uma função através dum seu parâmetro*
- Objetos instância
  - *são os objetos típicos de qualquer linguagem orientada a objetos, as instâncias das classes*
  - *permitem-nos, por exemplo, aceder aos seus membros na forma obj.x e obj.f(), sendo x um atributo e f() um método*
- Objetos função
  - *um método (ou função), sendo um objeto, pode ser atribuído a uma variável, e depois invocado a partir dessa variável*
  - *Exemplo:* `y = obj.f` *(sendo f() um método do objeto obj)*

    `y()` *(o mesmo que obj.f())*

Mas pronto, ainda que em Python seja tudo objeto, continuaremos, por norma, a referirmo-nos aos 'objetos instância', sempre que não especificarmos a natureza do objeto…

Encapsulamento

- O criador do Python decidiu não incluir os mecanismos de proteção que estão presentes noutras linguagens (como o C++, C# e Java), que permitem distinguir o que é público do que é protegido ou privado
  - *Assumindo que "somos todos adultos", deixa a responsabilidade ao programador de aceder unicamente àquilo que deverá ser a "interface" do objeto*
  - *Existem no entanto algumas convenções, ou seja, regras que não têm um carater impositivo nem impeditivo ([5]):*
    - prefixar com um *underscore* os atributos ou métodos que não queiramos que "sejam públicos" (_xpto)
      - *digamos que são membros que noutras linguagens classificaríamos como private ou protected*
    - não prefixar os nossos nomes com um duplo underscore (deixemos essa opção para os nomes predefinidos da linguagem)
      - *como já deve ter reparado, os nomes usados internamente pelo sistema começam e acabam por duplo underscore (__xpto__), para servirem de alerta*

        (tratam-se de palavras chave que não devem ser alteradas – não devem ser usadas para outro fim)

A cláusula *self nas classes Python*

class

- *uma classe é definida com a palavra chave 'class'*
- *todos os seus membros (atributos e métodos) devem formar um bloco indentado*

self

- *serve para os objetos se autorreferenciarem*
- *parecido com o 'this' do C++, Java e C#, mas com outra preponderância no Python*
- *vai ser determinante para se diferenciarem os membros (métodos e atributos) de objeto dos membros de classe*
  - por agora, saiba-se apenas que o Python converte invocações do tipo obj.metodo(a1,a2) em X.metodo(obj,a1,a2), sendo obj um objeto duma qualquer classe X.

  (chamar um método duma instância com n argumentos é equivalente a chamar a função correspondente da classe com esses n argumentos precedidos por um primeiro com a referência da instância – self)

Classes

- A classe mais simples possível é mostrada no seguinte exemplo

```
class Person:
    pass  # An empty block

p = Person()
print(p)
```

Output:

```
<__main__.Person instance at 0x10171f518>
```

  - *neste caso, a palavra 'pass' serve para indicar um bloco vazio*
    - que será o corpo da classe Person
  - *um objeto da classe Person é depois criado, passando a ficar referenciado pela variável p*
  - *um simples print do objeto mostra-nos: o módulo atual (__main__), a classe a que pertence o objeto (Person) e o endereço da sua localização na memória (0x10171f518)*

Métodos

- Vejamos o self em ação:

```
class Person:
    def say_hi(self):
        print('Hello, how are you?')

p = Person()
p.say_hi()
# The previous 2 lines can also be written as
# Person().say_hi()
```

Output:

```
Hello, how are you?
```

  - *Observe-se que o método say_hi() é invocado sem qualquer argumento mas, ainda assim, temos o parâmetro self na sua definição*
    - é a inclusão do self na definição que faz com que o método seja do objeto, e não da classe

Métodos inicializadores (construtores)

- Os métodos construtores em Python têm o nome __init__

  (em Java, C++ e C# têm o nome da própria classe, como se sabe)

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

    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Swaroop')
p.say_hi()
```

Output:

```
Hello, my name is Swaroop
```

Aqui, definimos o método __init__ com um parâmetro name (além do parâmetro self)

- *mas também criamos um novo atributo com o nome 'name', usando self como qualificador*
  - é também esse qualificador que o torna um atributo de objeto, e não de classe

- Em Python cada classe só tem um construtor
  - *ainda assim, diferentes formas de inicialização poderão ser asseguradas através de parâmetros opcionais*
- Por conseguinte, podemos assumir que há apenas 2 tipos de construtores em Python
  - *construtor por defeito – aquele que não tem parâmetros e que é gerado automaticamente sempre que não é definido explicitamente qualquer outro*
  - *construtor parametrizado – todos os que têm parâmetros.*

Outros métodos especiais

- Para além do __init__(self), existem outros métodos que têm um significado especial dentro de uma classe do Python
  - *São normalmente utilizados para emular comportamentos de certas funções ou operações que tenham como operando o objeto*
    - Por exemplo, se quisermos usar a indexação obj[key] com objetos da nossa classe (exatamente como se faz com as listas), então tudo o que é necessário fazer é implementar o método __getitem__(.)

      (é precisamente isso que o Python faz na classe *list*)
  - *E são normalmente invocados usando uma sintaxe especial (não usando, portanto, o seu nome de forma explícita)*
  - *É assim que o Python assegura a sobrecarga dos operadores*
- Apresentam-se de seguida alguns dos métodos especiais
  - *Para um conhecimento mais detalhado destes e de todos os outros métodos especiais, consultar:*

  https://docs.python.org/3/reference/datamodel.html#special-method-names

Alguns dos métodos especiais

- `__init__(self, ...)`

  É automaticamente executado no momento da criação do objeto – conhecido por método inicializador ou construtor.

- `__del__(self)`

  É automaticamente executado imediatamente antes do objeto ser eliminado – conhecido por método destrutor.

- `__str__(self)`

  Invocado quando usadas as funções print(obj) ou str(obj), sendo obj um dos objetos da classe.

- `__lt__(self, other)`

  Invocado quando usado o operador menor (obj<other, sendo obj e other objetos da classe).

  Similarmente, existem métodos análogos para todos os outros operadores (+, ==, >=, etc.)

- `__getitem__(self, key)`

  Invocado quando é usada a indexação obj[key], sendo obj um dos objetos da classe.

- `__len__(self)`

  Invocado quando usada a função len(obj), sendo obj um dos objetos da classe

Identidade vs Igualdade

- Considerem-se os seguintes objetos:
  - *x1=[1,2,3]; x2=[1,2,3] ; x3=x1;*
- Dois objetos podem ser iguais, mas não com a mesma identidade
  - *exemplo: x1 e x2*
- Já tendo a mesma identidade, são necessariamente iguais
  - *exemplo: x1 e x3*
- Vejamos como se comportam esses objetos com o operador igualdade
  - *x1==x2*

    True
  - *x1==x3*

    True
  - *x2==x3*

    True

e com o operador identidade

- *x1 is x2*

  False
- *x1 is x3*

  True
- *x2 is x3*

  False

- Por conseguinte,
  - *o operador == é equivalente ao método equals() do Java,*
  - *e o 'is' é equivalente ao operador == do Java*
- Quando se usa o operador igualdade está-se a invocar o método __eq__(self, outro)
  - *Ou seja, x1==x2 é o mesmo que x1.__eq__(x2)*
  - *Logo, se estiverem em causa objetos de classes nossas, o operador == só terá o comportamento adequado se definirmos devidamente o método __eq__(self, outro)*

    (precisamente o cuidado que tínhamos que ter com o método equals() do Java)

Iteradores

- Um iterador é um objeto que nos permite percorrer os elementos de uma coleção ou sequência, sem precisarmos de conhecer a organização interna dessas estruturas.
- Em Python, um iterador tem o método __next__(),
  - *que devolve um elemento diferente da coleção de cada vez que é chamado, de forma sequencial*
  - *e lança a exceção* StopIteration quando já não existem mais elementos
- Esse método e a exceção é tudo o que precisamos para aceder aos elementos da coleção, um a um,
  - tal como se ilustra no exemplo

  (veja-se a seguir como se obtém o objeto iterador it)

```
lista=[2, 5, 1, 3]
it=iter(lista)
while True:
    try:
        x=next(it) #x=it.__next__()
        print(x, end=' ')
    except StopIteration:
        break
```

```
2 5 1 3
```

Iteráveis

- Um iterável é um objeto (normalmente uma coleção ou sequência) que contém o método __iter__(),
  - *o qual devolve um objeto iterador*
- No exemplo anterior, a instrução iter(lista) invoca o método lista.__iter__()
- A iteração de uma coleção iterável ou sequência faz-se de uma forma mais cómoda usando a estrutura for
  - *Solução equivalente para o exemplo anterior, sem o uso explícito de iteradores:*

```
lista=[2, 5, 1, 3]
for x in lista:
    print(x, end=' ')
```

```
2 5 1 3
```

  - *Note-se, no entanto, que esta solução mais compacta acaba por ser traduzida pelo Python na iteração explícita correspondente, semelhante à mostrada anteriormente*
- A função range(), frequentemente usada no ciclo for, devolve precisamente um objeto iterável

```
for i in range(1,50,3):
    print(i, end=' ')
```

```
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49
```

Tornar os nossos objetos iteráveis

- Como tornar então um objeto iterável?
  - *Define-se na respetiva classe um método* __iter__(), *que devolva um objeto iterador*
    - i. e., que devolva um objeto que contenha o método __next()__
- Na verdade, em Python o próprio objeto iterador inclui, para além do método __next()__, o método __iter__()
  - *que faz dele também um objeto iterável*
  - *sendo por essa razão que se pode usar o for e o in, quer em objetos iteráveis, quer sobre os próprios iteradores*

  (o método __iter__() de um iterador limita-se a devolver self)
- Tornar uma classe iterável dá algum trabalho…
  - *envolve, para além da implementação do método __iter__() na classe que queremos iterável, a criação do objeto iterador, com o seu método __next()__*

Funções geradoras

- Felizmente, o Python (à semelhança do C#) disponibiliza-nos um mecanismo adicional que simplifica bastante a criação de iteradores – as funções geradoras
  - *Uma função geradora devolve um objeto especial (o objeto gerador) que se comporta como um iterador (é um tipo de iterador)*
  - *Nessa função só temos que indicar que valores queremos que sejam devolvidos sequencialmente pelo objeto gerador, um por cada invocação do seu método* __next__()
    - definimos então a função como se devolvesse vários valores, um de cada vez (um por cada invocação de método __next__() do objeto gerador)
    - é através da instrução especial yield que indicamos cada um dos valores a devolver, sem terminar a função (com a instrução return apenas seria devolvido o primeiro… ☹)
  - *Assim, a forma mais simples de tornar uma coleção iterável passa por implementar o seu método* __iter__() *como uma função geradora.* ☺

Gerar um iterador – exemplo

- No exemplo que se segue, implementa-se o método __iter__() da classe que se pretende iterável (Alunos), como uma função geradora
  - *logo, devolve um objeto gerador, que mais não é que um iterador*

```
class Alunos:
    def __init__(self, *nomes):
        self.nomes=nomes

    def __iter__(self):
        for nome in self.nomes:
            yield nome
```

- Sendo a classe alunos iterável, já a podemos percorrer com um ciclo for....

```
alunos=Alunos('Ana','Rita','Rui')

for a in alunos:
    print(a)
```

```
Ana
Rita
Rui
```