Capítulo 11

Programação com Objectos

1. Defina uma classe em Python, chamada `estacionamento`, que simula o funcionamento de um parque de estacionamento. A classe `estacionamento` recebe um inteiro que determina a lotação do parque e devolve um objeto com os seguintes métodos: `entra()`, corresponde à entrada de um carro; `sai()`, corresponde à saída de um carro; `lugares()` indica o número de lugares livres no estacionamento. Por exemplo,

```
>>> ist = estacionamento(20)
>>> ist.lugares()
20
>>> ist.entra()
>>> ist.entra()
>>> ist.entra()
>>> ist.entra()
>>> ist.sai()
>>> ist.lugares()
17
```

2. Suponha que desejava criar a classe *racional* em Python. Um número racional é qualquer número que possa ser expresso como o quociente de dois inteiros: o numerador (um inteiro positivo, negativo ou nulo) e o denominador (um inteiro positivo ou negativo). Os racionais a/b e c/d são iguais se e só se $a \times d = b \times c$. Assuma que a representação externa de um racional é apresentada de modo que o numerador e o denominador são primos entre si. A classe *racional* admite as operações *nume* e *deno* que devolvem, respetivamente o numerador e o denonimador.

   (a) Defina a classe *racional*, incluindo o transformador de saída.

   (b) Usando operações polimórficas, escreva métodos para calcular a soma e o produto de racionais. Se $r_1 = a/b$ e $r_2 = c/d$ então $r_1 + r_2 = (ad + bc)/bd$ e $r_1 * r_2 = (a * c)/(b * d)$. Por exemplo,

```
>>> r1 = racional(2, 4)
>>> r2 = racional(1, 6)
>>> r1
1/2
>>> r2
1/6
>>> r1 + r2
2/3
>>> r1*r2
1/2
```

3. Os automóveis mais recentes mostram a distância que é possível percorrer até ser necessário um reabastecimento. Pretende-se criar esta funcionalidade em Python através da classe `automovel`. Esta classe é construída indicando a capacidade do depósito, a quantidade de combustível no depósito e o consumo do automóvel em litros aos 100 km. A classe `automovel` apresenta os seguintes métodos:

   - `combustivel` devolve a quantidade de combustível no depósito;
   - `autonomia` devolve o numero de Km que é possível percorrer com o combustível no depósito;
   - `abastece(n_litros)` aumenta em `n_litros` o combustível no depósito. Se este abastecimento exceder a capacidade do depósito, gera um erro e não aumenta a quantidade de combustível no depósito;
   - `percorre(n_km)` percorre `n_km` Km, desde que a quantidade de combustível no depósito o permita, em caso contrário gera um erro e o trajecto não é efectuado.

   Por exemplo:

   ```
   >>> a1 = automovel(60, 10, 15)
   >>> a1.combustivel()
   10
   >>> a1.autonomia()
   66
   >>> a1.abastece(45)
   '366 Km até abstecimento'
   >>> a1.percorre(150)
   '216 Km até abstecimento'
   >>> a1.percorre(250)
   ValueError: Não tem autonomia para esta viagem
   ```

4. Suponha que desejava criar a classe *conjunto*. Considere as seguintes operações para conjuntos:

   Construtores:

- $conjunto : \{\} \mapsto conjunto$

  $conjunto()$ tem como valor um conjunto sem elementos.

- $insere : elemento \times conjunto \mapsto conjunto$

  $insere(e, c)$ tem como valor o resultado de inserir o elemento e no conjunto c; se e já pertencer a c, tem como valor c.

Seletores:

- $el\_conj : conjunto \mapsto elemento$

  $el\_conj(c)$ tem como valor um elemento escolhido aleatoriamente do conjunto c; se o conjunto for vazio esta operação é indefinida.

- $retira\_conj : elemento \times conjunto \mapsto conjunto$

  $retira\_conj(e, c)$ tem como valor o resultado de retirar do conjunto c o elemento e; se e não pertencer a c, tem como valor c.

- $cardinal : conjunto \mapsto inteiro$

  $cardinal(c)$ tem como valor o número de elementos do conjunto c.

Reconhecedores:

- $e\_conj\_vazio : conjunto \mapsto lógico$

  $e\_conj\_vazio(c)$ tem o valor $verdadeiro$ se o conjunto c é o conjunto vazio, e tem o valor $falso$, em caso contrário.

Testes:

- $pertence : elemento \times conjunto \mapsto lógico$

  $pertence(e, c)$ tem o valor $verdadeiro$ se o elemento e pertence ao conjunto c e tem o valor $falso$, em caso contrário.

Operações adicionais:

- $subconjunto : conjunto \times conjunto \mapsto lógico$

  $subconjunto(c_1, c_2)$ tem o valor $verdadeiro$, se o conjunto c_1 for um subconjunto do conjunto c_2, ou seja, se todos os elementos de c_1 pertencerem a c_2, e tem o valor $falso$, em caso contrário.

- $uniao : conjunto \times conjunto \mapsto conjunto$

  $uniao(c_1, c_2)$ tem como valor o conjunto união de c_1 com c_2, ou seja, o conjunto formado por todos os elementos que pertencem a c_1 ou a c_2.

- $interseccao : conjunto \times conjunto \mapsto conjunto$

  $interseccao(c_1, c_2)$ tem como valor o conjunto intersecção de c_1 com c_2, ou seja, o conjunto formado por todos os elementos que pertencem simultaneamente a c_1 e a c_2.

- $diferenca : conjunto \times conjunto \mapsto conjunto$
  $diferenca(c_1, c_2)$ tem como valor o conjunto diferença de c_1 e c_2, ou seja, o conjunto formado por todos os elementos que pertencem a c_1 e não pertencem a c_2.

Defina a classe `conjunto`.

5. Considere a função de Ackermann:

$$A(m,n) = \begin{cases} n+1 & \text{se } m = 0 \\ A(m-1,1) & \text{se } m > 0 \text{ e } n = 0 \\ A(m-1, A(m, n-1)) & \text{se } m > 0 \text{ e } n > 0 \end{cases}$$

esta função pode ser directamente implementada em Python através da função:

```
def A(m, n):
    if m == 0:
        return n + 1
    elif m > 0 and n == 0:
        return A(m-1, 1)
    else:
        return A(m-1, A(m, n-1))
```

Como pode verificar, esta função calcula várias vezes o mesmo valor. Para evitar este problema, podemos definir uma classe, `mem_A`, cujo estado interno contém informação sobre os valores de `A` já calculados, apenas calculando um novo valor quando este ainda não é conhecido. Esta classe possui um método `val` que calcula o valor de `A` para os inteiros que são seus argumentos e um método `mem` que mostra os valores memorizados. Por exemplo,

```
>>> a = mem_A()
>>> a.val(2, 3)
9
>>> a.mem()
{(0, 1): 2,
 (0, 2): 3,
 (0, 3): 4,
 (0, 4): 5,
 (0, 5): 6,
 (0, 6): 7,
 (0, 7): 8,
 (0, 8): 9,
 (1, 0): 2,
 (1, 1): 3,
 (1, 2): 4,
```

```
(1, 3): 5,
(1, 4): 6,
(1, 5): 7,
(1, 6): 8,
(1, 7): 9,
(2, 0): 3,
(2, 1): 5,
(2, 2): 7,
(2, 3): 9}
```

Defina a classe `mem_A`.