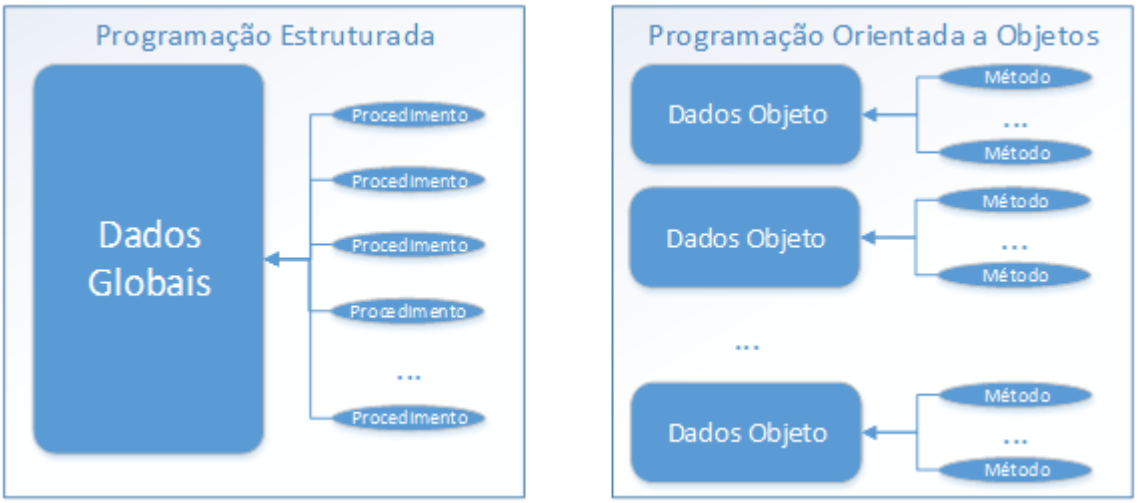
Introdução à programação orientada a objetos

A programação tradicional versus a programação orientada a objetos

Tradicionalmente a programação de sistemas considera os dados separados das funções. Os dados são estruturados de modo a facilitar a sua manipulação pelas funções, mas as funções estão livres para usar os dados como quiserem. Os aspectos segurança e integridade dos dados ficam de certa forma vulneráveis. A programação tradicional de sistemas tem o seu mais forte e bem-sucedido modelo na programação estruturada, utilizada largamente por linguagens como C, Pascal, Algol, etc.

Na programação orientada a objetos (POO), os dados específicos do objeto são estruturados juntamente com as funções que são permitidas sobre esses dados. Essa forma de programar veio com linguagens como Java, C++ e recentemente Python dentre outras.



fonte: http://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264

O elemento principal da POO são os objetos. Dizemos que um **objeto** é uma **instância de uma classe**. Uma Classe é constituída por variáveis (data members) e métodos ou funções (function members).
A Classe é o modelo. Um objeto é um elemento deste modelo.
As variáveis de uma Classe são também chamadas de Atributos.
As funções de uma Classe são também chamadas de Métodos.

Objetivos da POO:

1) Robustez – o programa não pode cair frente a dados inesperados.
2) Adaptabilidade – rodar facilmente em diferentes ambientes
3) Reusabilidade – usar os elementos já construídos em outros sistemas

Princípios da POO:

1) Modularidade – dividir o sistema em pequenas partes bem definidas.
2) Abstração – identificar as partes fundamentais (tipos de dados e operações), definindo o que cada operação faz e não necessariamente como é feito.
3) Encapsulamento – a interface para uso de cada componente deve estar bastante clara para todos que usam esse componente. Detalhes internos de implementação não interessam.

O desenvolvimento de software

Não há uma regra, método ou processo que oriente o desenvolvimento de bons programas e sistemas de computador. Há apenas diretrizes gerais que quando usadas levam em geral a um bom resultado. Além disso, o desenvolvimento certamente é influenciado pelo ambiente computacional no qual será desenvolvido. A linguagem de programação e o ambiente ou plataforma na qual será desenvolvido o sistema podem decidir o modo, a estratégia e os passos que serão usados.

Podemos dividir o desenvolvimento de software em 3 etapas, independente da linguagem, sistema operacional ou plataforma de desenvolvimento que será usada:

1) O projeto
2) A implementação
3) Os testes e depuração

O projeto é a parte mais importante. Nele são definidas as classes e a relação entre elas. Os princípios que devem orientar a definição das classes são:

a) Responsabilidade de cada uma delas – quais problemas elas resolve
b) Independência entre elas – se uma precisa da outra e vice-versa é melhor que sejam a mesma.
c) Comportamento – quais as entradas (parâmetros) e saídas (resultados) desta classe.

Antes do projeto há é claro, uma especificação de projeto fornecida. A qualidade desta especificação é fundamental para que o projeto atenda aos requisitos solicitados. É muito comum uma especificação sub ou superestimada o que vai implicar num projeto com os mesmos vícios.

É fundamental uma interação com os demandantes e prováveis futuros usuários do projeto para que a qualidade da especificação seja testada. Esse é aliás, um princípio que orienta recentes metodologias conhecidas como Metodologias Ágeis para o Desenvolvimento de Software (métodos SCRUM e XP de Extreme Programming). Em resumo o Manifesto Ágil, proposto por grupos de desenvolvedores adeptos desta metodologia em 2001, preconiza:

- Pessoas e interações - ao contrário de processos e ferramentas.
- Software executável - ao contrário de documentação extensa e confusa.
- Colaboração do cliente - ao contrário de constantes negociações de contratos.
- Respostas rápidas para as mudanças - ao contrário de seguir planos previamente definidos

Na fase de implementação, embora cada programador tenha seu estilo, é importante garantir a compatibilidade dentro da sua parte e entre as partes dos vários programadores do mesmo sistema.

A fase de testes e depuração é a que consome mais tempo. Muitas vezes, é durante a fase de testes que se descobre problemas do projeto e temos que retroceder à fase de projeto para corrigir e garantir a integridade do sistema.

Um exemplo de programação orientada a objeto

Supondo que temos que desenvolver um sistema que manipula certos produtos. O objeto em questão é o produto. Assim, vamos criar a classe Produto, que tornará possível construir ou criar elementos ou objetos desta classe.

classe: Produto
atributos: _nome, _codigo, _preco, _quantidade
métodos: obtem_nome, obtem_codigo, obtem_preco, altera_preco, altera_quantidade

Estamos usando uma convenção usual, onde os nomes dos identificadores ou variáveis internas (atributos) que caracterizam o objeto iniciam com underline. Não precisa ser assim. O nome pode ser qualquer.

```
class Produto:

    '''Define a classe Produto.'''

    # Construtor da classe Produto
    def __init__(self, nome, codigo, preco, quantidade):
        '''Cria uma instância de Produto.'''
        self._nome = nome
        self._codigo = codigo
        self._preco = preco
        self._quantidade = quantidade

    # Retorna com o nome do produto
    def obtem_nome(self):
        return self._nome

    # Retorna com o código do produto
    def obtem_codigo(self):
        return self._codigo

    # Retorna com o preço corrente do produto
    def obtem_preco(self):
        return self._preco

    # Devolve True se novo preço maior que o anterior
    def altera_preco(self, novo_preco):
        pp = self._preco
        self._preco = novo_preco
        if novo_preco > pp: return True
        return False

    # Devolve False se a quantidade de produtos requerida não está disponível
    def altera_quantidade(self, novo_pedido):
        if novo_pedido > self._quantidade: return False
        self._quantidade -= novo_pedido
        return True

# testes da classe

if __name__ == "__main__":
    p1 = Produto("Camisa Social", 123456, 45.56, 1000)
    p2 = Produto("Calça Jeans", 423564, 98.12, 500)

    print("Oferta do dia:", p1.obtem_nome())
    print("Oferta da semana:", p2.obtem_nome())

    # altera o preço
    if p1.altera_preco(40.00): print("Preço alterado hoje")
    else: print("Atenção - baixou o preço")

    # verifica se pode fazer uma venda
    if p2.altera_quantidade(100):
        print("Pode fazer a venda - valor total = ", p2.obtem_preco() * 100)
    else: print("Não tem produto suficiente para esta venda")
```

O nome self refere-se ao particular objeto sendo criado. Note que o primeiro parâmetro é sempre self na definição. No uso ou na chamada do método esse primeiro parâmetro não existe.

No exemplo acima incluímos além da definição da classe, alguns comandos de teste dos métodos da mesma. Assim o módulo (arquivo onde está armazenada a classe) poderia chamar-se ClasseProduto.py.

O comando if __name__ == "__main__" usado antes dos testes acima, determina que os comandos abaixo somente serão executados quando a classe estiver sendo testada, isto é, quando for solicitada a execução do módulo ClasseProduto.py.

Isso é bastante útil. Toda vez que uma classe é modificada, é necessário garantir que as modificações não introduziram erros. Testes de regressão tem que ser realizados para nos certificarmos disso. Portanto, ao deixarmos todos os testes anteriores dentre do módulo, será fácil realizar todos os testes novamente.

Como importar a classe já definida

Feita a classe Produto dentro do módulo ClasseProduto.py, ela pode ser importada (import) e usada por outros módulos. Para isso dentro do diretório em que está o módulo ClasseProduto.py, que é normalmente o diretório corrente, deve estar presente também um módulo vazio de nome __init__.py. Isso sinaliza ao sistema Python que classes podem ser importadas deste diretório.

Uso e declaração dos métodos

Quando o método é declarado sempre o primeiro parâmetro é self, quando é necessário acesso ao objeto. Entretanto nas chamadas omite-se esse parâmetro. O correspondente ao self torna-se o prefixo da chamada.

Declaração:
```
def altera_preco(self, novo_preco):
```

Uso:
```
if p1.altera_preco(40.00):
```

O método construtor

__init__ é um método especial dentro da classe. É construtor da classe, onde os atributos do objeto recebem seus valores iniciais. No caso da classe acima, os 4 atributos (variáveis) que caracterizam um produto, recebem neste método seus valores iniciais, que podem ser modificados por operações futuras deste mesmo objeto. A criação de um novo produto, ou seja, a criação de uma instância do objeto produto causa a execução do método __init__. Assim, o comando abaixo, causa a execução do método __init__:

```
p1 = Produto("Camisa Social", 123456, 45.56, 1000)
```

Robustez

Para que a nossa classe exemplo acima se torne mais robusta, é necessário incluir os testes de consistência dos parâmetros e eventualmente retornar códigos de erro e em alguns casos sinalizar exceção. Por exemplo, no caso de preço ou código negativo, nomes fora do padrão, quantidades nulas ou negativas, etc.
Numa classe real, todas essas situações precisam ser tratadas.

Explorando um pouco mais a sintaxe das classes

Abaixo um exemplo de uma classe simples para exemplificar mais a sintaxe. A classe Duplas tem apenas o método construtor, um método que soma duas duplas e um método que mostra a dupla.

```
class Duplas:

    '''Define a classe Duplas.'''

    # Construtor da classe
    def __init__(self, p1 = 0, p2 = 0):
        '''Construtor da classe.'''
        self.primeiro = p1
```

```
        self.segundo = p2

    # Retorna no primeiro parâmetro a soma de dois elementos da classe
    def SomaDuplas(self, d1, d2):
        self.primeiro = d1.primeiro + d2.primeiro
        self.segundo = d1.segundo + d2.segundo

    # Mostre uma dupla no formato /a b/
    def Mostre(self):
        print("/", self.primeiro, self.segundo, "/")

# Exemplos de uso da classe

# /3 4/
x = Duplas(3, 4)
x.Mostre()

# /2 0/
y = Duplas(2)
y.Mostre()

# /0 0/
z = Duplas()
z.Mostre()

# z = x + y
z.SomaDuplas(x, y)
z.Mostre()

# z = /5 8/ + /1 1/
z.SomaDuplas(Duplas(5, 8), Duplas(1, 1))
z.Mostre()
```

Será impresso:

```
/ 3 4 /
/ 2 0 /
/ 0 0 /
/ 5 4 /
/ 6 9 /
```

Abaixo algumas variações. A SomaDuplas() retorna elemento Duplas e o parâmetro de Mostre() não é self.

```
class Duplas:

    '''Define a classe Duplas.'''

    # Construtor da classe
    def __init__(self, p1 = 0, p2 = 0):
        '''Construtor da classe.'''
        self.primeiro = p1
        self.segundo = p2

    # Retorna a soma de dois elementos da classe
    # Neste caso, nem precisaria do parâmetro self
    def SomaDuplas(self, d1, d2):
```

```
        pri = d1.primeiro + d2.primeiro
        seg = d1.segundo + d2.segundo
        return Duplas(pri, seg)

    # Mostre uma dupla no formato / a b /
    def Mostre(param):
        print("/", param.primeiro, param.segundo, "/")

# Exemplos de uso da classe

# /3 4/
x = Duplas(3, 4)
x.Mostre() # x é o primeiro parâmetro de Mostre

# /2 0/
y = Duplas(2)
y.Mostre() # idem y

# /0 0/
z = Duplas()
z.Mostre() # idem z

# x + y - mostrado o resultado do return em SomaDuplas
Duplas().SomaDuplas(x, y).Mostre()

# /5 8/ + /1 1/ - note que z não é alterado
z.SomaDuplas(Duplas(5, 8), Duplas(1, 1))
z.Mostre()

# /5 8/ + /1 1/ - agora sim, z é alterado
z = Duplas().SomaDuplas(Duplas(5, 8), Duplas(1, 1))
z.Mostre()
```

Será impresso:

```
/ 3 4 /
/ 2 0 /
/ 0 0 /
/ 5 4 /
/ 0 0 /
/ 6 9 /
```

A função ou método dentro de uma classe tem um funcionamento um pouco diferente. Tanto o primeiro parâmetro (em geral o self) quanto os demais parâmetros formais, se alterados dentro da função, alteram também os respectivos parâmetros atuais.

No exemplo abaixo a função SomaDuplas() altera também os dois outros parâmetros.

```
class Duplas:

    '''Define a classe Duplas.'''

    # Construtor da classe
    def __init__(self, p1 = 0, p2 = 0):
        '''Construtor da classe.'''
        self.primeiro = p1
```

```
        self.segundo = p2

    # Retorna no primeiro parâmetro a soma de dois elementos da classe
    def SomaDuplas(self, d1, d2):
        x = d1.primeiro + d2.primeiro
        y = d1.segundo + d2.segundo
        self.primeiro = x
        self.segundo = y
        # Outras atribuições  - só como exemplo
        d1.primeiro = d2.primeiro = -1
        d1.segundo = d2.segundo = -1

    # Mostre uma dupla no formato / a b /
    def Mostre(self):
        print("/", self.primeiro, self.segundo, "/")

# Exemplos de uso da classe

# /3 4/
x = Duplas(3, 4)
x.Mostre()

# /2 0/
y = Duplas(2)
y.Mostre()

# /0 0/
z = Duplas()
z.Mostre()

# z = x + y - SomaDuplas() altera também x e y
z.SomaDuplas(x, y)
z.Mostre()
x.Mostre()
y.Mostre()
```

Será impresso:

```
/ 3 4 /
/ 2 0 /
/ 0 0 /
/ 5 4 /
/ -1 -1 /
/ -1 -1 /
```

Continuando a explorar a sintaxe das classes – o parâmetro self

O parâmetro self é importante no método construtor da classe __init__, para referenciar o objeto que está sendo criado. Nos demais métodos, se usado, é simplesmente um parâmetro. Nem precisa ser o primeiro. Mesmo no __init__, o primeiro parâmetro pode ter qualquer nome.

```
class com_init:

    def __init__(outro, valor):
        outro.val = valor
```

```
    def f1(a, b, self):
        print("entrou em f1:", a.val, b.val, self.val)

# testes da classe
x = com_init(-5)
y = com_init(-4)
z = com_init(-3)
x.f1(y, z)
```

Quem imprime:

```
entrou em f1: -5 -4 -3
```

Uma classe não precisa necessariamente ter um método construtor. Podemos ter uma classe apenas com métodos, sem atributos. Nesse caso, o nome da classe é usado sem parâmetros para a chamada das funções.

```
class sem_init:

    def f1(x, y):
        print("entrou em f1:", x, y)

    def f2(a, self, b):
        print("entrou em f2:", a, self, b)

# testes da classe
sem_init.f1(5,4)
sem_init.f2(5, 4, 3)
```

Quem imprime:

```
entrou em f1: 5 4
entrou em f2: 5 4 3
```

Métodos especiais e sobrecarga de operadores

Em Python o mesmo operador tem significados diferentes dependendo do tipo dos operandos.
Por exemplo, a + b é uma soma se a e b são numéricos, uma concatenação de a e b são strings, listas ou tuplas.

É possível criar operações para outros objetos, usando o mesmo operador. Existem métodos especiais que dão suporte à implementação de operadores com objetos de classes definidas pelo usuário.

Considere a classe Duplas definida anteriormente.

Seria muito interessante poder usar a expressão x + y em vez da SomaDuplas(x, y). O método especial __add__ () dentro da classe permite que isso seja feito.

Da mesma forma, seria interessante também usar print(x) em vez de x.Mostre(). A função print(), precisa transformar o objeto em string antes de imprimi-lo. Isso pode ser feito usando a função __str__ () dentro da classe.

Vejamos agora como fica a nova classe Duplas com essas melhorias:

```
class Duplas:

    '''Define a classe Duplas.'''
```

```
    # Construtor da classe
    def __init__(self, p1 = 0, p2 = 0):
        '''Construtor da classe.'''
        self.primeiro = p1
        self.segundo = p2

    # Retorna a soma de dois elementos da classe
    def __add__(d1, d2):
        pri = d1.primeiro + d2.primeiro
        seg = d1.segundo + d2.segundo
        return Duplas(pri, seg)

    # Transforma uma Dupla em string para imprimir no formato / a b /
    def __str__(d):
        return "/" + str(d.primeiro) + " " + str(d.segundo) + "/"

# Exemplos de uso da classe

# /3 4/
x = Duplas(3, 4)
print(x)

# /2 0/
y = Duplas(2)
print(y)

# /0 0/
z = Duplas()
print(z)

# atribuição de soma a uma dupla
z = x + y
print(z)

# qualquer expressão contendo só soma
z = x + x + y + y
print(z)

# inclusive com constantes tipo dupla
z = x + Duplas(-1, -3) + Duplas(5, -5) + y
print(z)

# imprimindo direto o resultado
print(z + y + x + Duplas(-3))
```

Será impresso:

```
/3 4/
/2 0/
/0 0/
/5 4/
/10 8/
/9 -4/
/11 0/
```

P3.1) Expandir a classe Duplas com as operações de subtração, multiplicação e divisão. Os métodos especiais que realizam isso são __sub__(), __mult__() e __div__().

Outra aplicação da sobrecarga de operadores é dar uma interpretação diferente a um operador para determinada classe. Suponha por exemplo, que precisamos de um novo tipo de inteiro cuja soma seja igual ao maior deles. A classe abaixo define esse novo tipo e a operação. Para diferenciar de um inteiro comum, vamos imprimi-lo com um caractere # à frente.

```
class outro_int:

    ''' redefine soma de int. '''

    def __init__(self, valor):
        # construtor da classe
        self.val = valor

    def __add__(b1, b2):
        if b1.val > b2.val: return b1
        return b2

    def __str__(b):
        return '#' + str(b.val)

# testes da classe
x = outro_int(3)
y = outro_int(5)
print(x)
print(y)
z = x + y
print(z)
print(x + y)
```

Será impresso:

```
#3
#5
#5
#5
```

O resultado seria o mesmo, usando o parâmetro self. Este faz o papel de primeiro parâmetro.

```
    def __add__(self, b):
        if self.val > b.val: return outro_int(self.val)
        return outro_int(b.val)

    def __str__(self):
        return '#' + str(self.val)
```

P3.2) Refazer classe outro_int acima com as operações de soma(o maior em módulo), subtração (o menor em módulo), multiplicação (a soma dos módulos) e divisão (módulo da subtração). Os métodos especiais que realizam isso são __add__(), __sub__(), __mult__() e __div__().

Os métodos especiais para a sobrecarga de operadores

São vários os métodos especiais usados para redefinir operações de novos objetos.
A tabela abaixo mostra tais métodos.

fonte: [1] - Data Structures and Algorithms in Python – Goodrich, Tamassia, Goldwasser [DSA – GTG]

Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a * b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ** b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)
a += b a −= b a *= b ...	a.__iadd__(b) a.__isub__(b) a.__imul__(b) ...	
+a	a.__pos__()	
−a	a.__neg__()	
~a	a.__invert__()	
abs(a)	a.__abs__()	
a < b	a.__lt__(b)	
a <= b	a.__le__(b)	
a > b	a.__gt__(b)	
a >= b	a.__ge__(b)	
a == b	a.__eq__(b)	
a != b	a.__ne__(b)	
v in a	a.__contains__(v)	
a[k]	a.__getitem__(k)	
a[k] = v	a.__setitem__(k,v)	
del a[k]	a.__delitem__(k)	
a(arg1, arg2, ...)	a.__call__(arg1, arg2, ...)	
len(a)	a.__len__()	
hash(a)	a.__hash__()	
iter(a)	a.__iter__()	
next(a)	a.__next__()	
bool(a)	a.__bool__()	
float(a)	a.__float__()	
int(a)	a.__int__()	
repr(a)	a.__repr__()	
reversed(a)	a.__reversed__()	
str(a)	a.__str__()	

Considere a operação de soma (+). É com o método __add__() que o Python implementa as versões da soma para diferentes objetos. Inclusive para os tipos básicos (int, float, bool, strings, tupla e lista).

A expressão **a + b** torna-se de fato o seguinte comando **a.__add__(b)**.
É também desta forma que expressões do tipo **5 * 'exemplo'** são implementadas. Neste caso usando o operador __mul__. Como há mistura de operadores, é avaliada a conveniência de usar __mul__. Se não comporta a mistura, outro método é chamado __rmul__ que deve tratar a expressão, invertendo os operandos.

Processo similar ocorre quando usamos um método especial para tratar objetos de classes definidas pelo usuário. Podemos usar **str(obj)**, **int(obj)**, **len(obj)**, etc. usando as mesmas funções intrínsecas, mesmo quando **obj** é de classes definidas pelo usuário. Neste caso, o construtor da classe do método especial irá fazer uma chamada para classe a qual pertence esse objeto. Assim, **str(obj)** transforma-se em **obj.__str__()**, **int(obj)** em **obj.__int__()**, **len(obj)** em **obj.__len()__**.

Um exemplo de uso de sobrecarga de operadores – Vetores multidimensionais

Uma classe que realiza a soma e comparações com vetores da álgebra. Podemos usar uma lista para armazenar um vetor. Por exemplo, para dimensão 3, v1 = [2, -3, 9], v2 = [-1, 3, 5], v1 + v2 = [1, 0, 14].
Vamos então considerar os seguintes métodos:
__len__, __getitem__, __setitem__, __add__, __eq__, __ne__

```
# Exemplo baseado em [1]
class Vector:

    '''Vetor multidimensional.'''

    def __init__ (self, d):
        '''Cria vetor de dimensão d com zeros.'''
        self._coords = [0] * d

    def __len__ (self):
        '''Retorna a dimensão do vetor.'''
        return len(self._coords)

    def __getitem__ (self, j):
        '''Retorna a j-ésima coordenada.'''
        return self._coords[j]

    def __setitem__ (self, j, val):
        '''Define a j-ésima coordenada do vetor.'''
        self._coords[j] = val

    def __add__ (self, other):
        '''Retorna a soma de 2 vetores.'''
        if len(self) != len(other): # usa a função len desta classe
            raise ValueError("dimensões devem ser a mesma")
        result = Vector(len(self))
        for j in range(len(self)):
            result[j] = self[j] + other[j]
        return result

    def __eq__ (self, other):
        '''Retorna True se vetores iguais.'''
        return self._coords == other._coords

    def __ne__ (self, other):
        '''Retorna True se vetores são diferentes.'''
        return not self == other # usa a definição de eq
```

```
    def imprime (self):
        '''Imprime o vetor.'''
        print(self._coords)

# testes
v1 = Vector(3)
v2 = Vector(3)
v1[0] = 2 # setitem
v2[2] = 3 # setitem
v3 = v1 + v2 # add
if v1 == v3 or v2 == v3: # eq
    print("soma igual a um deles")
else:
    print("soma diferente")
if v3 != Vector(3): # ne
    print("Não é zero")
v3.imprime()
```

Herança de classes

Permite que uma classe seja definida com base em classe já existente. Dizemos que essa nova classe herda características da classe original. Na terminologia Python a classe original é chamada de Classe Base, Classe Mãe ou Superclasse (Base, Parent ou Super Class) enquanto que a nova é chamada de Sub Classe ou Classe Filha (Sub ou Child Class).

A subclasse pode **especializar** a classe principal ou mesmo **estendê-la** com novos métodos e atributos.

Como um exemplo, vamos estender a classe Produto, construindo a classe Produto Crítico. Esta nova classe possui um parâmetro a mais que é o estoque mínimo para esses produtos críticos. O estoque não pode ficar abaixo deste limite. Uma venda que invada esse limite é rejeitada. Outra característica destes produtos críticos é não permitir alteração de preço maior que 10% acima ou abaixo.

Assim, faremos a nova __init__ com um parâmetro a mais e reescrevemos as funções altera_quantidade e altera_preco. Não é preciso reescrevê-las completamente. Podemos chamar a função correspondente da classe mãe quando necessário.

```
class ProdutoCritico(Produto):
    '''Define a classe Produto Crítico.'''
    # Construtor da classe ProdutoCritico
    def __init__(self, nome, codigo, preco, quantidade, estoque_min):
        '''Cria uma instância de Produto Crítico.'''
        # Chama a __init__ da classe mãe
        super().__init__(nome, codigo, preco, quantidade)
        self._estoque_min = estoque_min

    # idem a classe mãe considerando estoque mínimo
    def altera_quantidade(self, novo_pedido):
        if novo_pedido + self._estoque_min > self._quantidade:
            return False
        return super().altera_quantidade(novo_pedido)

    # idem a classe mãe - False se alterou mais de 10%
    def altera_preco(self, novo_preco):
        pp = self._preco
        if novo_preco > 1.1 * pp or novo_preco < 0.9 * pp:
            return False
        ap = super().altera_preco(novo_preco)
        return True
```

```
# testes da nova classe ProdutoCritico

    pc1 = ProdutoCritico("Camiseta Social", 2123456, 5.56, 1000, 100)
    pc2 = ProdutoCritico("Calção Jeans", 2423564, 8.12, 500, 100)

    print("Oferta do dia:", pc1.__obtem_nome__())
    print("Oferta da semana:", pc2.__obtem_nome__())

    # altera o preço
    if pc1.altera_preco(4.00): print("Preço alterado hoje")
    else: print("Atenção - diferença > 10% - Preço não alterado")

    # verifica se pode fazer uma venda
    if pc2.altera_quantidade(800):
        print("Pode fazer a venda - valor total = ", pc2.obtem_preco() * 100)
    else: print("Não tem produto suficiente ou excedeu o estoque mínimo")
```

E será impresso:

```
Oferta do dia: Camiseta Social
Oferta da semana: Calção Jeans
Atenção - diferença > 10% - Preço não alterado
Não tem produto suficiente ou excedeu o estoque mínimo
```

No caso acima, ambas as classes, mãe e filha podem ser instanciadas. Não precisa ser necessariamente assim. A classe mãe pode ser apenas uma classe de referência para que as filhas herdem suas funções. Assim, poderíamos em vez de produtos críticos, podíamos dividir nossos produtos em vários tipos e criar uma classe para cada tipo. Cada tipo com eventuais atributos e funcionalidades diferentes. Exemplo:

class ProdutoOrganico
class ProdutoNaoOrganico
class ProdutoLimpeza
class ProdutoLaticinio
etc.

Assim, todas essas classes herdariam as características da classe Produto. Sempre a instância seria de uma classe filha, uma vez que qualquer produto seria de um dos tipos.

Escopo dos nomes (NameSpace) – Dados próprios da classe

Uma classe possui atributos e métodos. Quando uma instância é criada, um espaço de nomes é criado para os seus atributos (variáveis). Os métodos residem em outro espaço de nomes. Não há necessidade de recriar novas cópias dos métodos para cada instância, pois são os mesmos. Apenas referenciam variáveis diferentes para cada instância.

A classe pode também conter dados que podem ser acessados por todos os seus métodos. É um tipo de dado global à classe. Tipicamente constantes importantes à classe ficam mais bem referenciadas com nomes. No caso da classe Produto podemos considerar um estoque mínimo “default” para cada produto, ou um preço para cada produto.

```
class Produto:
    '''Define a classe Produto.'''
    PRECO_MINIMO = 1.00
    ESTOQUE_MINIMO_POSSIVEL = 5
    ...
```

Classes internas ou encaixadas

Uma classe pode ter outras classes internas a si.

```
class A:
    class B:
   ...
...
```

Fica claro que o nome B faz parte do escopo de nomes de A e pode ser usado por A, mas não pode ser usada por outras classes no mesmo nível que A. A vantagem é que em B podemos ter nomes que se replicam em outras classes sem conflito de nomes. Não deve ser confundida com a herança de classes explicada anteriormente.

A declaração __slots__

Para manter as variáveis de uma classe, Python usa uma instância da classe dicionário (dict) que tem um tamanho considerável para otimizar a busca. Isso consome memória. Para usar exatamente a quantidade necessária de memória, usa-se a declaração __slots__, definindo assim os nomes das variáveis que serão usadas. Se houver classes filhas, estas também terão que ter a declaração apenas com as novas variáveis. Na classe Produto ficaria:

```
class Produto:
    '''Define a classe Produto.'''

    # Declaração __slots__
    __slots__ = 'nome', 'codigo', 'preco', 'quantidade'
    . . . .

class ProdutoCritico(Produto):
    '''Define a classe Produto Crítico.'''

    # Declaração __slots__
    __slots__ = 'estoque_min'
    . . . .
```

Resolução dinâmica de nomes

Quando um atributo ou método é referenciado usando o ponto (.) como em:

```
pc1.__obtem_nome__())
pc1.altera_preco(4.00)
pc2.altera_quantidade(800):
```

A busca do nome para identificar onde está o atributo ou a função segue alguns passos dentre os possíveis espaços de nomes existentes:

1) procura nos nomes da instância desse objeto
2) se não encontrou, procura na classe a qual pertence esse objeto
3) se não encontrou, procura hierarquicamente nas classes herdadas por esse objeto

Fica claro então que nos exemplos acima, as funções altera_preco e altera_quantidade da classe ProdutoCritico, se sobrepõe as funções de mesmo nome na classe Produto.

Essa busca pelo nome é feita pelo Python de forma dinâmica, em tempo de execução.

Cópias e sinônimos de listas

Já vimos que comandos de atribuição do tipo x = y, apenas criam sinônimos, ou seja, x e y referenciam o mesmo objeto. Isso deve ser levado em conta se x e y são objetos mais complexos, compostos por objetos mutáveis. Quando for necessário obtermos uma cópia real de um objeto, funções especiais tem que ser usadas.

Abaixo damos exemplos usando listas como objeto. O mesmo se aplica para novos objetos definidos.

```
x = [32, 45]
y = x
x[0] = -1
print(x, y)
print(id(x), id(y))
```

Imprime:

```
[-1, 45] [-1, 45]
48024912 48024912
```

A forma seguinte resolve parcialmente o problema de tornar x e y elementos diferentes:

```
x = [32, 45]
y = list(x)
x[0] = -1
print(x, y)
print(id(x), id(y))
```

Que imprime:

```
 [-1, 45] [32, 45]
48575752 48251128
```

Esse tipo de cópia é chamado de cópia leve (shallow copy). Dizemos que resolve parcialmente porque uma nova instância da lista é criada, mas referenciando os mesmos objetos em memória. Isso traz um problema adicional se os objetos forem mutáveis. No exemplo abaixo, alteramos x e y também foi alterado. Isso não ocorre se x e y forem imutáveis.

```
x = [[32, 45], [81, 37, 75]]
y = list(x)
x[0][0] = -1
print(x, y)
print(id(x), id(y))
```

Imprime:

```
[[-1, 45], [81, 37, 75]] [[-1, 45], [81, 37, 75]]
48351592 48480504
```

Outras formas de obter cópias de listas

Há outras de fazer cópias de listas. Veja abaixo.
Entretanto, quando a lista é composta por objetos mutáveis, por exemplo uma lista de listas o efeito acima sempre ocorre.

```
a = [[1, 2], [3, 4, 5], [6, 7, 8, 9]]

# 1
# cria b com o mesmo tamanho de a
```

```
b = [[0] for k in range(len(a))]
# copia elemento por elemento
for k in range(len(a)): b[k] = a[k]

# 2
b = []
# b precisa existir para que o comando abaixo funcione
b[:] = a[:]

# 3
# nesse caso, será criado um novo objeto
b = a[:]

# 4
# nesse caso, b é o resultado de uma concatenação
b = [] + a
```

Não ocorre o mesmo quando listas com mesmo conteúdo são criadas independentemente.

```
mx = [[2, 3], [4, 5, 6], [7, 8, 9, 0]]
my = [[2, 3], [4, 5, 6], [7, 8, 9, 0]]
mz = [[2, 3], [4, 5, 6], [7, 8, 9, 0]]

m1 = [m * [0] for k in range(n)]
my = [m * [0] for k in range(n)]
```

A forma de evitar esse efeito com listas que possuem elementos mutáveis é fazer uma cópia real da mesma usando a função deepcopy abaixo.

Cópias definitivas - deepcopy

O modulo copy, permite que sejam feitas cópias reais (deep copy). Possui duas funções, a copy que faz uma cópia leve (shallow copy) e deepcopy que faz uma cópia real (deep copy). Nesse último caso, além da nova instância, um novo conjunto de dados é criado. Necessário importar o módulo copy (import copy).

```
x = [[32, 45], [81, 37, 75]]
y = copy.deepcopy(x)
x[0][0] = -1
print(x, y)
print(id(x), id(y))
```

Que imprime:

```
[[-1, 45], [81, 37, 75]] [[32, 45], [81, 37, 75]]
48154864 48155024
```