CAPÍTULO 09

9. Classes

Classes proporcionam uma forma de organizar dados e funcionalidades juntos. Criar uma nova classe cria um novo “tipo” de objeto, permitindo que novas “instâncias” desse tipo sejam produzidas. Cada instância da classe pode ter atributos anexados a ela, para manter seu estado. Instâncias da classe também podem ter métodos (definidos pela classe) para modificar seu estado.

Em comparação com outras linguagens de programação, o mecanismo de classes de Python introduz a programação orientada a objetos sem acrescentar muitas novidades de sintaxe ou semântica. É uma mistura de mecanismos equivalentes encontrados em C++ e Modula-3. As classes em Python oferecem todas as características tradicionais da programação orientada a objetos: o mecanismo de herança permite múltiplas classes base (herança múltipla), uma classe derivada pode sobrescrever quaisquer métodos de uma classe ancestral, e um método pode invocar outro método homônimo de uma classe ancestral. Objetos podem armazenar uma quantidade arbitrária de dados de qualquer tipo. Assim como acontece com os módulos, as classes fazem parte da natureza dinâmica de Python: são criadas em tempo de execução, e podem ser alteradas após sua criação.

Usando a terminologia de C++, todos os membros de uma classe (incluindo dados) são *públicos* (veja exceção abaixo Variáveis privadas), e todos as funções membro são *virtuais*. Como em Modula-3, não existem atalhos para referenciar membros do objeto de dentro dos seus métodos: o método (função definida em uma classe) é declarado com um primeiro argumento explícito representando o objeto (instância da classe), que é fornecido implicitamente pela chamada ao método. Como em Smalltalk, classes são objetos. Isso fornece uma semântica para importar e renomear. Ao contrário de C++ ou Modula-3, tipos pré-definidos podem ser utilizados como classes base para extensões por herança pelo usuário. Também, como em C++, a maioria dos operadores (aritméticos, indexação, etc) podem ser redefinidos por instâncias de classe.

(Na falta de uma terminologia universalmente aceita para falar sobre classes, ocasionalmente farei uso de termos comuns em Smalltalk ou C++. Eu usaria termos de Modula-3, já que sua semântica de orientação a objetos é mais próxima da de Python, mas creio que poucos leitores já ouviram falar dessa linguagem.)

9.1. Uma palavra sobre nomes e objetos

Objetos têm individualidade, e vários nomes (em diferentes escopos) podem ser vinculados a um mesmo objeto. Isso é chamado de apelidamento em outras linguagens. Geralmente, esta característica não é muito apreciada, e pode ser ignorada com segurança ao lidar com tipos imutáveis (números, strings, tuplas). Entretanto, apelidamento pode ter um efeito surpreendente na semântica do código Python envolvendo objetos mutáveis como listas, dicionários e a maioria

dos outros tipos. Isso pode ser usado em benefício do programa, porque os apelidos funcionam de certa forma como ponteiros. Por exemplo, passar um objeto como argumento é barato, pois só um ponteiro é passado na implementação; e se uma função modifica um objeto passado como argumento, o invocador verá a mudança — isso elimina a necessidade de ter dois mecanismos de passagem de parâmetros como em Pascal.

9.2. Escopos e espaços de nomes do Python

Antes de introduzir classes, é preciso falar das regras de escopo em Python. Definições de classe fazem alguns truques com espaços de nomes. Portanto, primeiro é preciso entender claramente como escopos e espaços de nomes funcionam, para entender o que está acontecendo. Esse conhecimento é muito útil para qualquer programador Python avançado.

Vamos começar com algumas definições.

Um *espaço de nomes* é um mapeamento que associa nomes a objetos. Atualmente, são implementados como dicionários em Python, mas isso não é perceptível (a não ser pelo desempenho), e pode mudar no futuro. Exemplos de espaços de nomes são: o conjunto de nomes pré-definidos (funções como `abs()` e as exceções pré-definidas); nomes globais em um módulo; e nomes locais na invocação de uma função. De certa forma, os atributos de um objeto também formam um espaço de nomes. O mais importante é saber que não existe nenhuma relação entre nomes em espaços de nomes distintos. Por exemplo, dois módulos podem definir uma função de nome `maximize` sem confusão — usuários dos módulos devem prefixar a função com o nome do módulo, para evitar colisão.

A propósito, utilizo a palavra *atributo* para qualquer nome depois de um ponto. Na expressão `z.real`, por exemplo, `real` é um atributo do objeto `z`. Estritamente falando, referências para nomes em módulos são atributos: na expressão `modname.funcname`, `modname` é um objeto módulo e `funcname` é um de seus atributos. Neste caso, existe um mapeamento direto entre os atributos de um módulo e os nomes globais definidos no módulo: eles compartilham o mesmo espaço de nomes! [1]

Atributos podem ser somente leitura ou para leitura e escrita. No segundo caso, é possível atribuir um novo valor ao atributo. Atributos de módulos são passíveis de atribuição: você pode escrever `modname.the_answer = 42`. Atributos que aceitam escrita também podem ser apagados através da instrução `del`. Por exemplo, `del modname.the_answer` removerá o atributo `the_answer` do objeto referenciado por `modname`.

Espaços de nomes são criados em momentos diferentes e possuem diferentes ciclos de vida. O espaço de nomes que contém os nomes embutidos é criado quando o interpretador inicializa e nunca é removido. O espaço de nomes global de um módulo é criado quando a definição do módulo é lida, e normalmente duram até a terminação do interpretador. Os comandos executados pela

invocação do interpretador, pela leitura de um script com programa principal, ou interativamente, são parte do módulo chamado __main__, e portanto possuem seu próprio espaço de nomes. (Os nomes embutidos possuem seu próprio espaço de nomes no módulo chamado builtins.).

O espaço de nomes local de uma função é criado quando a função é invocada, e apagado quando a função retorna ou levanta uma exceção que não é tratada na própria função. (Na verdade, uma forma melhor de descrever o que realmente acontece é que o espaço de nomes local é "esquecido" quando a função termina.) Naturalmente, cada invocação recursiva de uma função tem seu próprio espaço de nomes.

Um *escopo* é uma região textual de um programa Python onde um espaço de nomes é diretamente acessível. Aqui, "diretamente acessível" significa que uma referência sem um prefixo qualificador permite o acesso ao nome.

Ainda que escopos sejam determinados estaticamente, eles são usados dinamicamente. A qualquer momento durante a execução, existem 3 ou 4 escopos aninhados cujos espaços de nomes são diretamente acessíveis:

- o escopo mais interno, que é acessado primeiro, contem os nomes locais
- os escopos das funções que envolvem a função atual, que são acessados a partir do escopo mais próximo, contém nomes não-locais, mas também não-globais
- o penúltimo escopo contém os nomes globais do módulo atual
- e o escopo mais externo (acessado por último) contém os nomes das funções embutidas e demais objetos pré-definidos do interpretador

Se um nome é declarado no escopo global, então todas as referências e atribuições de valores vão diretamente para o escopo intermediário, que contém os nomes globais do módulo. Para alterar variáveis declaradas fora do escopo mais interno, a instrução nonlocal pode ser usada; caso contrário, todas essas variáveis serão apenas para leitura (a tentativa de atribuir valores a essas variáveis simplesmente criará uma *nova* variável local, no escopo interno, não alterando nada na variável de nome idêntico fora dele).

Normalmente, o escopo local referencia os nomes locais da função corrente no texto do programa. Fora de funções, o escopo local referencia os nomes do escopo global: espaço de nomes do módulo. Definições de classes adicionam um outro espaço de nomes ao escopo local.

É importante perceber que escopos são determinados estaticamente, pelo texto do código-fonte: o escopo global de uma função definida em um módulo é o espaço de nomes deste módulo, sem importar de onde ou por qual apelido a função é invocada. Por outro lado, a busca de nomes é dinâmica, ocorrendo durante a execução. Porém, a evolução da linguagem está caminhando para uma resolução de nomes estática, em "tempo de compilação", portanto não conte com a resolução dinâmica de nomes! (De fato, variáveis locais já são resolvidas estaticamente.)

Uma peculiaridade especial do Python é que – se nenhuma instrução global ou nonlocal estiver em vigor – as atribuições de nomes sempre entram no escopo mais interno. As atribuições não copiam dados — elas apenas vinculam nomes aos objetos. O mesmo vale para exclusões: a instrução `del x` remove a ligação de `x` do espaço de nomes referenciado pelo escopo local. De fato, todas as operações que introduzem novos nomes usam o escopo local: em particular, instruções import e definições de funções ligam o módulo ou o nome da função no escopo local.

A instrução global pode ser usada para indicar que certas variáveis residem no escopo global ao invés do local; a instrução nonlocal indica que variáveis particulares estão em um espoco mais interno e devem ser recuperadas lá.

9.2.1. Exemplo de escopos e espaço de nomes

Este é um exemplo que demonstra como se referir aos diferentes escopos e aos espaços de nomes, e como global e nonlocal pode afetar ligação entre as variáveis:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

A saída do código de exemplo é:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
```

```
In global scope: global spam
```

Observe como uma atribuição *local* (que é o padrão) não altera o vínculo de *scope_test* a *spam*. A instrução nonlocal mudou o vínculo de *scope_test* de *spam* e a atribuição global alterou a ligação para o nível do módulo.

Você também pode ver que não havia nenhuma ligação anterior para *spam* antes da atribuição global.

9.3. Uma primeira olhada nas classes

Classes introduzem novidades sintáticas, três novos tipos de objetos, e também alguma semântica nova.

9.3.1. Sintaxe da definição de classe

A forma mais simples de definir uma classe é:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Definições de classe, assim como definições de função (instruções def), precisam ser executadas antes que tenham qualquer efeito. (Você pode colocar uma definição de classe dentro do teste condicional de um if ou dentro de uma função.)

Na prática, as instruções dentro da definição de classe geralmente serão definições de funções, mas outras instruções são permitidas, e às vezes são bem úteis — voltaremos a este tema depois. Definições de funções dentro da classe normalmente têm um forma peculiar de lista de argumentos, determinada pela convenção de chamada a métodos — isso também será explicado mais tarde.

Quando se inicia a definição de classe, um novo espaço de nomes é criado, e usado como escopo local — assim, todas atribuições a variáveis locais ocorrem nesse espaço de nomes. Em particular, funções definidas aqui são vinculadas a nomes nesse escopo.

Quando uma definição de classe é finalizada normalmente (até o fim), um *objeto classe* é criado. Este objeto encapsula o conteúdo do espaço de nomes criado pela definição da classe; aprenderemos mais sobre objetos classe na próxima seção. O escopo local que estava vigente antes da definição da classe é

reativado, e o objeto classe é vinculado ao identificador da classe nesse escopo (ClassName no exemplo).

9.3.2. Objetos de Class

Objetos classe suportam dois tipos de operações: *referências a atributos* e *instanciação*.

Referências a atributos de classe utilizam a sintaxe padrão utilizada para quaisquer referências a atributos em Python: obj.nome. Nomes de atributos válidos são todos os nomes presentes dentro do espaço de nomes da classe, quando o objeto classe foi criado. Portanto, se a definição de classe tem esta forma:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

então MyClass.i e MyClass.f são referências a atributo válidas, retornando, respectivamente, um inteiro e um objeto função. Atributos de classe podem receber valores, pode-se modificar o valor de MyClass.i num atribuição. __doc__ também é um atributo válido da classe, retornando a *documentação* associada: "A simple example class".

Para *instanciar* uma classe, usa-se a mesma sintaxe de invocar uma função. Apenas finja que o objeto classe do exemplo é uma função sem parâmetros, que devolve uma nova instância da classe. Por exemplo (assumindo a classe acima):

```
x = MyClass()
```

cria uma nova *instância* da classe e atribui o objeto resultante à variável local x.

A operação de instanciação (“invocar” um objeto classe) cria um objeto vazio. Muitas classes preferem criar novos objetos com um estado inicial predeterminado. Para tanto, a classe pode definir um método especial chamado __init__(), assim:

```
def __init__(self):
    self.data = []
```

Quando uma classe define um método __init__(), o processo de instanciação automaticamente invoca __init__() sobre a instância recém criada. Em nosso exemplo, uma nova instância já inicializada pode ser obtida desta maneira:

```
x = MyClass()
```

Naturalmente, o método __init__() pode ter parâmetros para maior flexibilidade. Neste caso, os argumentos fornecidos na invocação da classe serão passados para o método __init__(). Por exemplo,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. Objetos instância

Agora o que podemos fazer com objetos de instância? As únicas operações compreendidas por objetos de instância são os atributos de referência. Existem duas maneiras válidas para nomear atributos: atributos de dados e métodos.

Atributos de dados correspondem a “variáveis de instância” em Smalltalk, e a “membros de dados” em C++. Atributos de dados não precisam ser declarados. Assim como variáveis locais, eles passam a existir na primeira vez em que é feita uma atribuição. Por exemplo, se `x` é uma instância da `MyClass` criada acima, o próximo trecho de código irá exibir o valor `16`, sem deixar nenhum rastro:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

O outro tipo de referências a atributos de instância é o “método”. Um método é uma função que “pertence” a um objeto instância. (Em Python, o termo método não é aplicado exclusivamente a instâncias de classes definidas pelo usuário: outros tipos de objetos também podem ter métodos. Por exemplo, listas possuem os métodos append, insert, remove, sort, entre outros. Porém, na discussão a seguir, usaremos o termo método apenas para se referir a métodos de classes definidas pelo usuário. Seremos explícitos ao falar de outros métodos.)

Nomes de métodos válidos de uma instância dependem de sua classe. Por definição, cada atributo de uma classe que é uma função corresponde a um método das instâncias. Em nosso exemplo, `x.f` é uma referência de método válida já que `MyClass.f` é uma função, enquanto `x.i` não é, já que `MyClass.i` não é

uma função. Entretanto, `x.f` não é o mesmo que `MyClass.f`. A referência `x.f` acessa um objeto método e a `MyClass.f` acessa um objeto função.

9.3.4. Objetos método

Normalmente, um método é chamado imediatamente após ser referenciado:

```
x.f()
```

No exemplo `MyClass` o resultado da expressão acima será a string `'hello world'`. No entanto, não é obrigatório invocar o método imediatamente: como `x.f` é também um objeto ele pode ser atribuído a uma variável e invocado depois. Por exemplo:

```
xf = x.f
while True:
    print(xf())
```

exibirá o texto `hello world` até o mundo acabar.

O que ocorre precisamente quando um método é invocado? Você deve ter notado que `x.f()` foi chamado sem nenhum argumento, porém a definição da função `f()` especificava um argumento. O que aconteceu com esse argumento? Certamente Python levanta uma exceção quando uma função que declara um argumento é invocada sem nenhum argumento — mesmo que o argumento não seja usado no corpo da função…

Na verdade, pode-se supor a resposta: a particularidade sobre os métodos é que o objeto da instância é passado como o primeiro argumento da função. Em nosso exemplo, a chamada `x.f()` é exatamente equivalente a `MyClass.f(x)`. Em geral, chamar um método com uma lista de *n* argumentos é equivalente a chamar a função correspondente com uma lista de argumentos que é criada inserindo o objeto de instância do método antes do primeiro argumento.

Se você ainda não entende como os métodos funcionam, dê uma olhada na implementação para esclarecer as coisas. Quando um atributo de uma instância, não relacionado a dados, é referenciado, a classe da instância é pesquisada. Se o nome é um atributo de classe válido, e é o nome de uma função, um método é criado, empacotando a instância e a função, que estão juntos num objeto abstrato: este é o método. Quando o método é invocado com uma lista de argumentos, uma nova lista de argumentos é criada inserindo a instância na posição 0 da lista. Finalmente, o objeto função — empacotado dentro do objeto método — é invocado com a nova lista de argumentos.

9.3.5. Variáveis de classe e instância

De forma geral, variáveis de instância são variáveis que indicam dados que são únicos a cada instância individual, e variáveis de classe são variáveis de atributos e de métodos que são comuns a todas as instâncias de uma classe:

```
class Dog:

    kind = 'canine'           # class variable shared by all
instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each
instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

Como vimos em Uma palavra sobre nomes e objetos, dados compartilhados podem causar efeitos inesperados quando envolvem objetos (mutáveis), como listas ou dicionários. Por exemplo, a lista *tricks* do código abaixo não deve ser usada como variável de classe, pois assim seria compartilhada por todas as instâncias de *Dog*:

```
class Dog:

    tricks = []             # mistaken use of a class variable

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

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
```

```
['roll over', 'play dead']
```

Em vez disso, o modelo correto da classe deve usar uma variável de instância:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each
dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4. Observações aleatórias

Se um mesmo nome de atributo ocorre tanto na instância quanto na classe, a busca pelo atributo prioriza a instância:

```
>>> class Warehouse:
        purpose = 'storage'
        region = 'west'

>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

Atributos de dados podem ser referenciados por métodos da própria instância, bem como por qualquer outro usuário do objeto (também chamados “clientes” do objeto). Em outras palavras, classes não servem para implementar tipos puramente abstratos de dados. De fato, nada em Python torna possível assegurar o encapsulamento de dados — tudo é baseado em convenção. (Por

outro lado, a implementação de Python, escrita em C, pode esconder completamente detalhes de um objeto e controlar o acesso ao objeto, se necessário; isto pode ser utilizado por extensões de Python escritas em C.)

Clientes devem utilizar atributos de dados com cuidado, pois podem bagunçar invariantes assumidas pelos métodos ao esbarrar em seus atributos de dados. Note que clientes podem adicionar atributos de dados a suas próprias instâncias, sem afetar a validade dos métodos, desde que seja evitado o conflito de nomes. Novamente, uma convenção de nomenclatura poupa muita dor de cabeça.

Não existe atalho para referenciar atributos de dados (ou outros métodos!) de dentro de um método. Isso aumenta a legibilidade dos métodos: não há como confundir variáveis locais com variáveis da instância quando lemos rapidamente um método.

Frequentemente, o primeiro argumento de um método é chamado `self`. Isso não passa de uma convenção: o identificador `self` não é uma palavra reservada nem possui qualquer significado especial em Python. Mas note que, ao seguir essa convenção, seu código se torna legível por uma grande comunidade de desenvolvedores Python e é possível que alguma *IDE* dependa dessa convenção para analisar seu código.

Qualquer objeto função que é atributo de uma classe, define um método para as instâncias dessa classe. Não é necessário que a definição da função esteja textualmente embutida na definição da classe. Atribuir um objeto função a uma variável local da classe é válido. Por exemplo:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Agora `f`, `g` e `h` são todos atributos da classe `C` que referenciam funções, e consequentemente são todos métodos de instâncias da classe `C`, onde `h` é exatamente equivalente a `g`. No entanto, essa prática serve apenas para confundir o leitor do programa.

Métodos podem invocar outros métodos usando atributos de método do argumento `self`:

```
class Bag:
```

```
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Métodos podem referenciar nomes globais da mesma forma que funções comuns. O escopo global associado a um método é o módulo contendo sua definição na classe (a classe propriamente dita nunca é usada como escopo global!). Ainda que seja raro justificar o uso de dados globais em um método, há diversos usos legítimos do escopo global. Por exemplo, funções e módulos importados no escopo global podem ser usados por métodos, bem como as funções e classes definidas no próprio escopo global. Provavelmente, a classe contendo o método em questão também foi definida neste escopo global. Na próxima seção veremos razões pelas quais um método pode querer referenciar sua própria classe.

Cada valor é um objeto e, portanto, tem uma *classe* (também chamada de *tipo*). Ela é armazenada como `object.__class__`.

9.5. Herança

Obviamente, uma característica da linguagem não seria digna do nome “classe” se não suportasse herança. A sintaxe para uma classe derivada é assim:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

O identificador `BaseClassName` deve estar definido no escopo que contém a definição da classe derivada. No lugar do nome da classe base, também são aceitas outras expressões. Isso é muito útil, por exemplo, quando a classe base é definida em outro módulo:

```
class DerivedClassName(modname.BaseClassName):
```

A execução de uma definição de classe derivada procede da mesma forma que a de uma classe base. Quando o objeto classe é construído, a classe base é lembrada. Isso é utilizado para resolver referências a atributos. Se um atributo requisitado não for encontrado na classe, ele é procurado na classe base. Essa

regra é aplicada recursivamente se a classe base por sua vez for derivada de outra.

Não há nada de especial sobre instanciação de classes derivadas: `DerivedClassName()` cria uma nova instância da classe. Referências a métodos são resolvidas da seguinte forma: o atributo correspondente é procurado através da cadeia de classes base, e referências a métodos são válidas se essa procura produzir um objeto função.

Classes derivadas podem sobrescrever métodos das suas classes base. Uma vez que métodos não possuem privilégios especiais quando invocam outros métodos no mesmo objeto, um método na classe base que invoca um outro método da mesma classe base pode, efetivamente, acabar invocando um método sobreposto por uma classe derivada. (Para programadores C++ isso significa que todos os métodos em Python são realmente `virtuais`.)

Um método sobrescrito em uma classe derivada, de fato, pode querer estender, em vez de simplesmente substituir, o método da classe base, de mesmo nome. Existe uma maneira simples de chamar diretamente o método da classe base: apenas chame `BaseClassName.methodname(self, arguments)`. Isso é geralmente útil para os clientes também. (Note que isto só funciona se a classe base estiver acessível como `BaseClassName` no escopo global).

Python tem duas funções embutidas que trabalham com herança:

- Use `isinstance()` para verificar o tipo de uma instância: `isinstance(obj, int)` será `True` somente se `obj.__class__` é a classe `int` ou alguma classe derivada de `int`.
- Use `issubclass()` para verificar herança entre classes: `issubclass(bool, int)` é `True` porque `bool` é uma subclasse de `int`. Porém, `issubclass(float, int)` é `False` porque `float` não é uma subclasse de `int`.

9.5.1. Herança múltipla

Python também suporta uma forma de herança múltipla. Uma definição de classe com várias classes bases tem esta forma:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Para a maioria dos casos mais simples, pense na pesquisa de atributos herdados de uma classe pai como o primeiro nível de profundidade, da esquerda para a direita, não pesquisando duas vezes na mesma classe em que há uma

sobreposição na hierarquia. Assim, se um atributo não é encontrado em DerivedClassName, é procurado em Base1, depois, recursivamente, nas classes base de Base1, e se não for encontrado lá, é pesquisado em Base2 e assim por diante.

De fato, é um pouco mais complexo que isso; a ordem de resolução de métodos muda dinamicamente para suportar chamadas cooperativas para super(). Essa abordagem é conhecida em outras linguagens de herança múltipla como chamar-o-próximo-método, e é mais poderosa que a chamada à função super, encontrada em linguagens de herança única.

A ordenação dinâmica é necessária porque todos os casos de herança múltipla exibem um ou mais relacionamentos de diamante (em que pelo menos uma das classes pai pode ser acessada por meio de vários caminhos da classe mais inferior). Por exemplo, todas as classes herdam de object, portanto, qualquer caso de herança múltipla fornece mais de um caminho para alcançar object. Para evitar que as classes base sejam acessadas mais de uma vez, o algoritmo dinâmico lineariza a ordem de pesquisa, de forma a preservar a ordenação da esquerda para a direita, especificada em cada classe, que chama cada pai apenas uma vez, e que é monotônica (significando que uma classe pode ser subclassificada sem afetar a ordem de precedência de seus pais). Juntas, essas propriedades tornam possível projetar classes confiáveis e extensíveis com herança múltipla. Para mais detalhes, veja https://www.python.org/download/releases/2.3/mro/.

9.6. Variáveis privadas

Variáveis de instância “privadas”, que não podem ser acessadas, exceto em métodos do próprio objeto, não existem em Python. No entanto, existe uma convenção que é seguida pela maioria dos programas em Python: um nome prefixado com um sublinhado (por exemplo: `_spam`) deve ser tratado como uma parte não-pública da API (seja uma função, um método ou um atributo de dados). Tais nomes devem ser considerados um detalhe de implementação e sujeito a alteração sem aviso prévio.

Uma vez que existe um caso de uso válido para a definição de atributos privados em classes (especificamente para evitar conflitos com nomes definidos em subclasses), existe um suporte limitado a identificadores privados em classes, chamado *desfiguração de nomes*. Qualquer identificador no formato `__spam` (pelo menos dois sublinhados no início, e no máximo um sublinhado no final) é textualmente substituído por `_classname__spam`, onde `classname` é o nome da classe atual com sublinhado(s) iniciais omitidos. Essa desfiguração independe da posição sintática do identificador, desde que ele apareça dentro da definição de uma classe.

A desfiguração de nomes é útil para que subclasses possam sobrescrever métodos sem quebrar invocações de métodos dentro de outra classe. Por exemplo:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update    # private copy of original update()
method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

O exemplo acima deve funcionar mesmo se MappingSubclass introduzisse um identificador __update uma vez que é substituído por _Mapping__update na classe Mapping e _MappingSubclass__update na classe MappingSubclass, respectivamente.

Note que as regras de desfiguração de nomes foram projetadas para evitar acidentes; ainda é possível acessar ou modificar uma variável que é considerada privada. Isso pode ser útil em certas circunstâncias especiais, como depuração de código.

Código passado para exec() ou eval() não considera o nome da classe que invocou como sendo a classe corrente; isso é semelhante ao funcionamento da instrução global, cujo efeito se aplica somente ao código que é compilado junto. A mesma restrição se aplica às funções getattr(), setattr() e delattr(), e quando acessamos diretamente o __dict__ da classe.

9.7. Curiosidades e conclusões

Às vezes, é útil ter um tipo semelhante ao “record” de Pascal ou ao “struct” de C, para agrupar alguns itens de dados. Uma definição de classe vazia funciona bem para este fim:

```
class Employee:
    pass
```

```
john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Um trecho de código Python que espera um tipo de dado abstrato em particular, pode receber, ao invés disso, uma classe que imita os métodos que aquele tipo suporta. Por exemplo, se você tem uma função que formata dados obtidos de um objeto do tipo "arquivo", pode definir uma classe com métodos read() e readline() que obtém os dados de um "buffer de caracteres" e passar como argumento.

Métodos de instância tem atributos também: m.__self__ é o objeto instância com o método m(), e m.__func__ é o objeto função correspondente ao método.

ATIVIDADES

1. **Classe Bola:** Crie uma classe que modele uma bola:

a. Atributos: Cor, circunferência, material
b. Métodos: trocaCor e mostraCor

2. **Classe Quadrado:** Crie uma classe que modele um quadrado:

a. Atributos: Tamanho do lado
b. Métodos: Mudar valor do Lado, Retornar valor do Lado e calcular Área;

3. **Classe Bomba de Combustível:** Faça um programa completo utilizando classes e métodos que:

a. Possua uma classe chamada bombaCombustível, com no mínimo esses atributos:

1. tipoCombustivel.
2. valorLitro
3. quantidadeCombustivel

b. Possua no mínimo esses métodos:

1. abastecerPorValor( ) – método onde é informado o valor a ser abastecido e mostra a quantidade de litros que foi colocada no veículo
2. abastecerPorLitro( ) – método onde é informado a quantidade em litros de combustível e mostra o valor a ser pago pelo cliente.
3. alterarValor( ) – altera o valor do litro do combustível.
4. alterarCombustivel( ) – altera o tipo do combustível.
5. alterarQuantidadeCombustivel( ) – altera a quantidade de combustível restante na bomba.

OBS: Sempre que acontecer um abastecimento é necessário atualizar a quantidade de combustível total na bomba.

4. **Classe Ponto e Retangulo:** Faça um programa completo utilizando funções e classes que:
    a. Possua uma classe chamada Ponto, com os atributos x e y.
    b. Possua uma classe chamada Retangulo, com os atributos largura e altura.
    c. Possua uma função para imprimir os valores da classe Ponto
    d. Possua uma função para encontrar o centro de um Retângulo.
    e. Você deve criar alguns objetos da classe Retangulo.
    f. Cada objeto deve ter um vértice de partida, por exemplo, o vértice inferior esquerdo do retângulo, que deve ser um objeto da classe Ponto.
    g. A função para encontrar o centro do retângulo deve retornar o valor para um objeto do tipo ponto que indique os valores de x e y para o centro do objeto.
    h. O valor do centro do objeto deve ser mostrado na tela
    i. Crie um menu para alterar os valores do retângulo e imprimir o centro deste retângulo.