Informação de Licenciamento

Porções dessas leituras foram modificadas ou traduzidas literalmente do ótimo livro Think Python 2e (http://greenteapress.com/wp/think-python-2e/) por Allen Downey (http://www.allendowney.com/wp/) e de materiais desenvolvidos para o curso 6.145 (MIT) desenvolvido por Adam Hartz (mailto:hz@mit.edu).

Essas notas são um trabalho em progresso! Se você tem perguntas, comentários ou sugestões, por favor poste-as no Piazza ou mande um email para `armelin@mit.edu` .

---

0) Introdução

Ao longo das últimas leituras, apresentamos o poder de *definição de funções*, que nos deu o poder de criar funções que poderíamos então abstrair e tratar como se tivessem sido "embutidas" no Python. Da mesma forma que poderíamos usar os recursos integrados de, por exemplo, adição ou subtração, poderíamos criar novas variações dessas operações e usá-las e tratá-las como se tivessem sido definidas como parte do Python.

Neste conjunto de leituras, veremos como fazer a mesma coisa não com procedimentos, mas com dados. Assim como o Python veio com várias operações e funções integradas que poderíamos construir usando a definição de função, Python também vem com vários tipos de dados integrados (muitos dos quais você já tem experiência): `int` , `float` , `str` , `list` , `dict` e alguns outros. Neste conjunto de leituras, exploraremos um meio de criar tipos personalizados de objetos.

1) Classes e Instâncias

No primeiro conjunto de leituras, apresentamos *objetos* como as "coisas" principais com as quais programas Python trabalham e notamos que cada objeto tem um *tipo* e um *valor*: o valor de um objeto determina a coisa exata que ele representa, e seu tipo determina os tipos de coisas que os programas podem fazer com ele (define o conjunto de operações válidas nesse tipo de objeto).

Ocasionalmente, usaremos terminologia diferente: podemos nos referir ao tipo de um objeto como sua *classe* e podemos dizer que o objeto em si é uma *instância* dessa classe.

Por exemplo, `int` é uma classe e alguns exemplos de instâncias dessa classe são: `478` , `1` e `3` . Da mesma forma, `str` é uma classe e alguns exemplos de instâncias dessa classe são: `"sandwich"` , `"1234"` , e `"name"` .

Em virtude de serem membros da mesma classe de objetos, podemos operar em qualquer string exatamente da mesma maneira, independentemente do valor particular que uma instância representa; por exemplo: podemos concatenar strings, podemos usar `len` para calcular seu comprimento, podemos fazer um loop sobre elas com `for` e podemos indexar nelas para encontrar os caracteres em locais específicos dentro delas, e podemos convertê-las para minúsculas com `x.lower()` . O tipo do objeto é o que determina as operações possíveis ao lidar com aquele objeto.

Neste conjunto de leituras, generalizaremos essa ideia e apresentaremos a ideia de criar nossos próprios tipos (ou classes) de objetos. Descobriremos que, depois de fazer isso, seremos capazes de tratar esses tipos de dados personalizados como se tivessem sido integrados ao Python.

Ao longo desse conjunto de leituras, será importante traçar uma distinção entre *criar* uma classe de objetos e *fazer uma instância* dessa classe. Quando falamos sobre a criação de uma classe, estamos falando sobre a definição de um novo tipo de objeto (incluindo como eles são representados internamente e também as operações que são possíveis para aquela classe de objetos); quando falamos sobre fazer uma *instância* de uma classe, estamos falando sobre fazer um objeto específico. Por exemplo, `list` é uma classe, e quando digitamos `[1, 2, 3, 4]` em Python, estamos criando uma instância dessa classe.

2) Classes e Atributos Personalizados

Até agora, falamos sobre isso de forma bastante abstrata; vamos começar a trabalhar e criar nossa primeira classe. Vamos imaginar que estamos escrevendo um programa para realizar alguns cálculos geométricos no plano.

Na notação matemática, os pontos são frequentemente escritos entre parênteses com uma vírgula separando as coordenadas. Por exemplo, $(0, 0)$ representa a origem e (x, y) representa o ponto x unidades à direita e y unidades acima da origem.

Existem várias maneiras de representar pontos em Python:

- Poderíamos armazenar as coordenadas separadamente em duas variáveis, `x` e `y`.
- Podemos armazenar as coordenadas como elementos em uma lista ou tupla.
- Poderíamos criar um novo tipo para representar pontos como objetos. De certa forma, criar um novo tipo é mais complicado do que as outras opções, mas tem vantagens que veremos em breve.

Um tipo definido pelo programador também é chamado de *classe* e é definido usando (talvez sem surpresa) a palavra-chave `class`. Nossa definição de primeira classe é assim:

```
class Point:
    pass
```

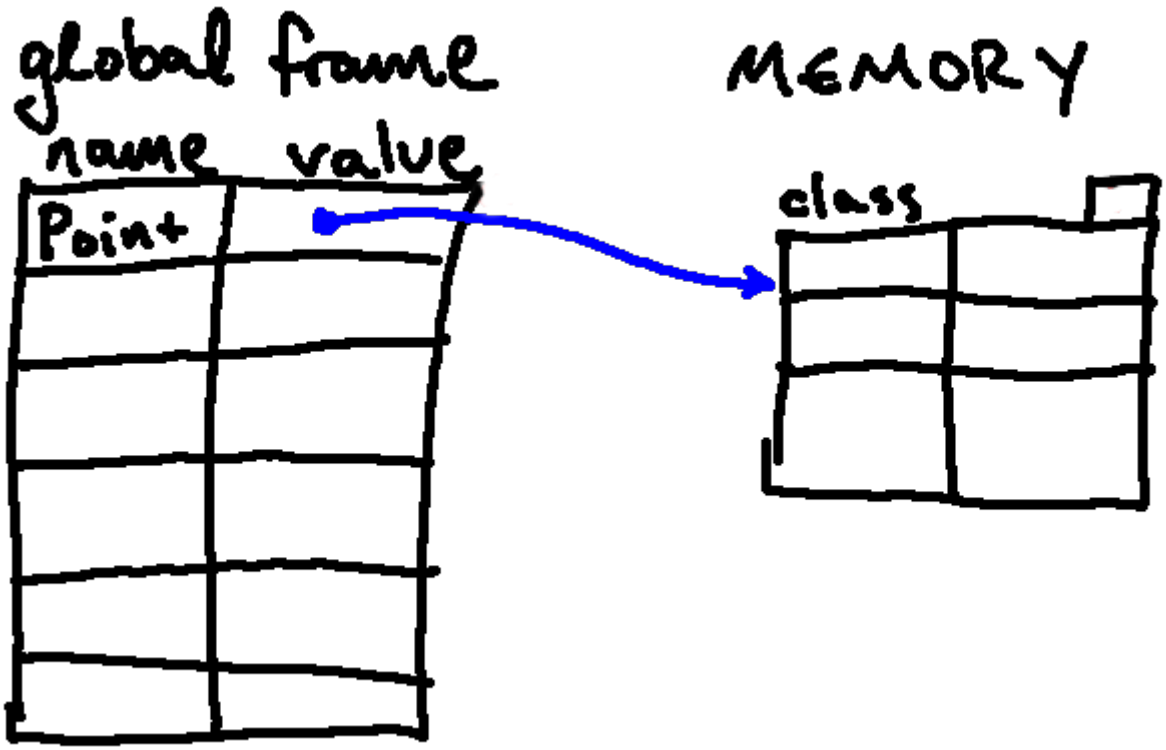
As definições de classe sempre começam com a palavra-chave `class`, seguida por um nome para a classe (aqui, `Point`), seguida por um corpo indentado.

Por enquanto, o corpo desta definição de classe não é particularmente interessante: consiste em uma única instrução para Python, `pass` (que é uma instrução Python que significa *não faça nada*). Você pode definir variáveis e funções dentro de uma definição de classe, mas vamos voltar a isso mais tarde.

Executar essa definição faz com que o Python faça duas coisas: primeiro, ele cria um *objeto de classe* para representar essa classe; e, em segundo lugar, associa o nome `Point` a este objeto de classe no quadro onde a classe foi definida.

Como sempre, precisaremos encontrar uma maneira de representar essas coisas novas em nossos diagramas de ambiente. Vamos representar objetos de classe de maneira semelhante a como representamos quadros ou dicionários, mas com uma nota para indicar que eles são de fato classes.

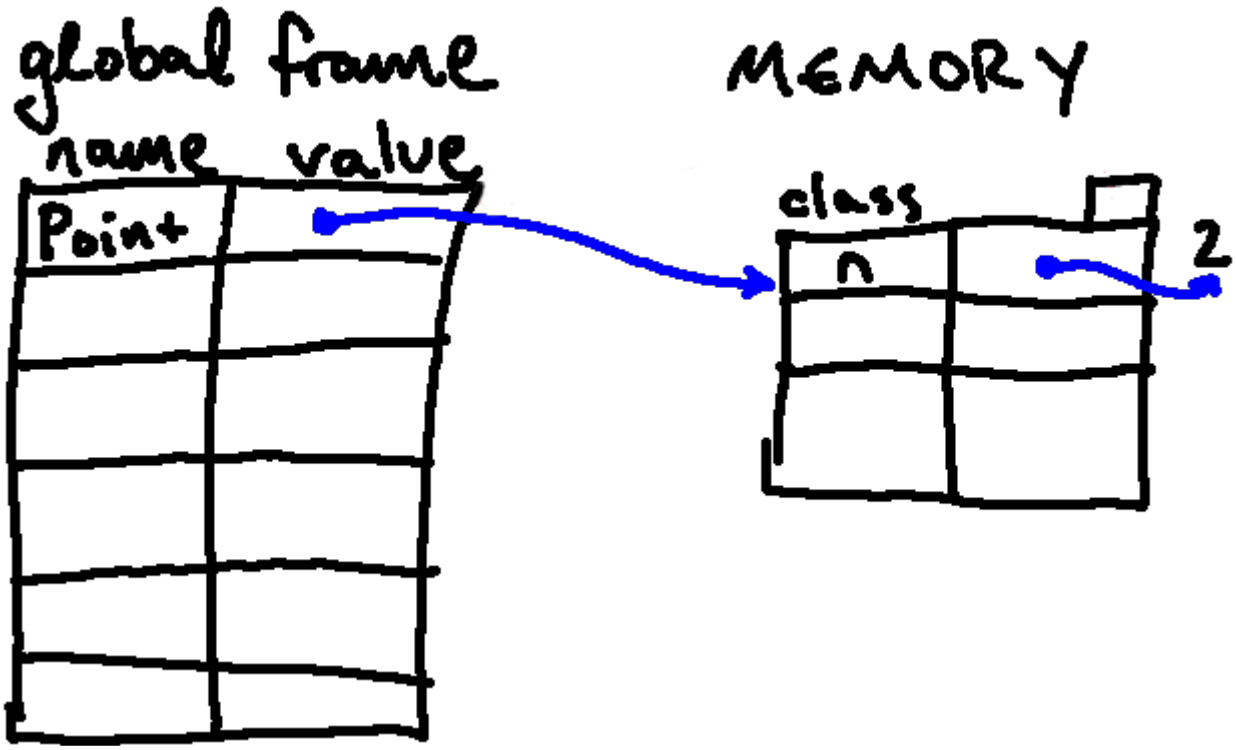
Avaliando a definição de classe acima nos deixa com o seguinte diagrama de ambiente (observe que esta instrução criou o objeto de classe e associou o nome `Point` a ele):



Quando o objeto de classe é criado, Python executará o corpo da classe *dentro desse ambiente*. É possível definir variáveis dentro da definição de classe (normalmente, para coisas que são comuns a todas as instâncias de uma classe). Por exemplo, se estivermos considerando apenas pontos no plano, poderíamos ter escrito:

```
class Point:
    # usando um nome de variável curto para representar o número de
    # coordenadas para que eu possa encaixá-las no diagrama de ambiente
    n = 2
```

o que resultaria no seguinte diagrama de ambiente:



Podemos procurar e/ou modificar atributos dentro de uma classe usando a mesma notação de ponto que usamos para módulos. Por exemplo, podemos usar o seguinte:

```
print(Point.n)
```

Para avaliar essa expressão, Python primeiro procura o nome `Point` (encontrando o objeto de classe) e, em seguida, procura o nome `n` dentro desse objeto, encontrando o inteiro `2`. Também podemos realizar atribuições usando notação semelhante:

```
Point.n = 3  # substitui a variável n dentro da classe
```

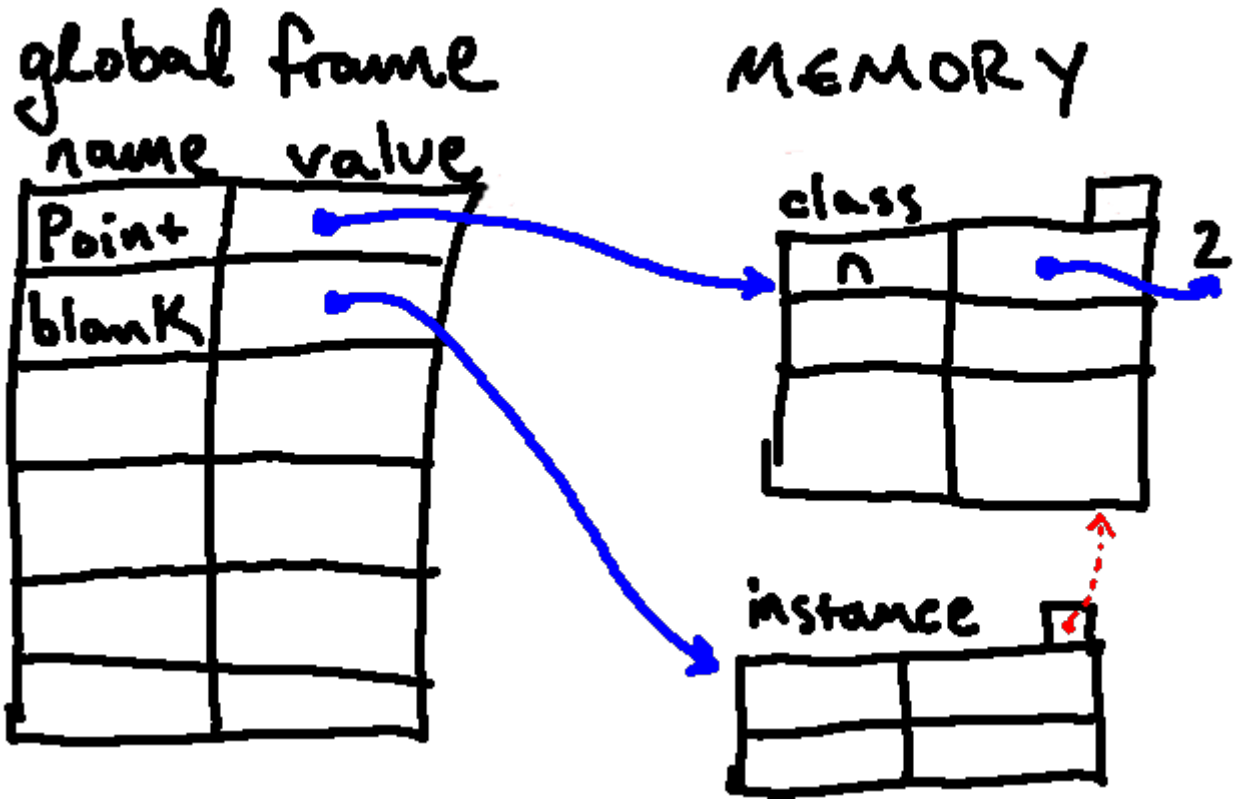
2.1) Criação de instâncias de classes definidas pelo usuário

O objeto de classe é como uma fábrica para a criação de objetos. Para criar um Ponto, você chama `Point` como se fosse uma função.

```
blank = Point()
```

O valor de retorno é uma referência a um objeto `Point`, que atribuímos a `blank`. A criação de um novo objeto é chamada de *instanciação* e o objeto é uma *instância* da classe.

Representaremos instâncias de classes definidas pelo usuário de maneira semelhante, exceto que as rotularemos como instâncias e seus ponteiros-pai apontarão de volta para sua classe associada. Portanto, avaliar a linha acima resultaria no seguinte diagrama de ambiente:



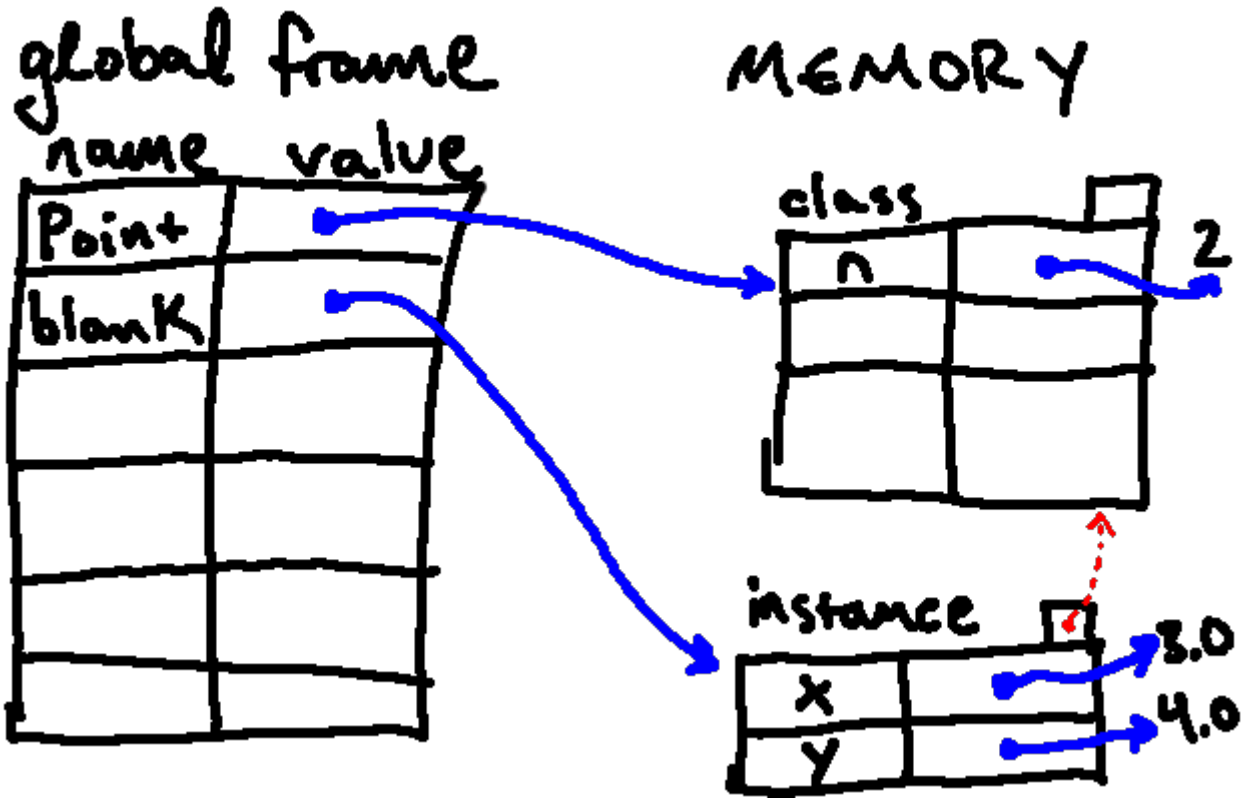
2.2) Atributos

Você pode atribuir valores a uma instância de uma classe definida pelo usuário usando a notação de ponto, da mesma forma que você os atribuiria dentro de uma classe:

```
blank.x = 3.0
blank.y = 4.0
```

Essa sintaxe é semelhante à sintaxe para selecionar uma variável de um módulo, como `math.pi` ou `string.whitespace`. Nesse caso, porém, estamos atribuindo valores a elementos nomeados de um objeto. Esses elementos são chamados de *atributos* ou *variáveis de instância*.

O diagrama a seguir mostra o resultado da execução dessas instruções:



A variável `blank` se refere a um objeto `Point`, que contém dois atributos. Cada atributo se refere a um float.

Você pode ler o valor de um atributo usando a mesma sintaxe:

```
print(blank.y)  # exibe 4.0
x = blank.x
print(x)  # exibe 3.0
```

Como vimos acima com classes, a expressão `blank.x` significa, "Procure o nome `blank` no quadro atual e procure o nome `x` dentro desse objeto". No exemplo, atribuímos esse valor a uma variável chamada `x`. Não há conflito entre a variável `x` e o atributo `x` porque um é definido no quadro global, mas o outro é definido dentro da instância.

Você pode usar a notação de ponto como parte de qualquer expressão. Por exemplo:

```
print('(', blank.x, ',', blank.y, ')')  # exibe (3.0, 4.0)

import math
distance = math.sqrt(blank.x ** 2 + blank.y ** 2)
print(distance)  # imprime 5.0
```

2.2.1) Resolução de Nome

Observe que, quando criamos nossa instância, demos a ela um ponteiro pai para a classe na qual foi definida. Com a sintaxe `blank.x`, procuramos o valor do nome `x` dentro da instância que criamos. Mas o que acontece se tentarmos pesquisar um nome que não existe?

Tente agora:

O que acontece quando você procura `blank.a`? Que tal `blank.n`?
Mostrar/Esconder

Pesquisar `blank.a` fornece um novo tipo de erro: um `AttributeError`, dizendo que este objeto não tem o atributo `a`. Procurar `blank.n`, por outro lado, nos dá `2`.

Se o Python estiver procurando dentro de um objeto por um atributo e não puder encontrá-lo, ele procurará esse nome em seguida na classe desse objeto. Se não o encontrar lá, ele fornecerá um `AttributeError` (observe que ele *não* continuará olhando além da classe; ou seja, não procurará no quadro global).

2.3) Exemplo: Retângulos

Às vezes é óbvio quais devem ser os atributos de um objeto, mas outras vezes você tem que tomar decisões. Por exemplo, imagine que você está projetando uma classe para representar retângulos. Que atributos você usaria para especificar a localização e o tamanho de um retângulo? Você pode ignorar o ângulo; para manter as coisas simples, suponha que o retângulo seja vertical ou horizontal.

Existem pelo menos duas possibilidades:

- Você pode especificar um canto do retângulo (ou o centro), a largura e a altura.
- Você pode especificar dois cantos opostos. Neste ponto, é difícil dizer se um é melhor do que o outro, então implementaremos o primeiro, apenas como um exemplo.

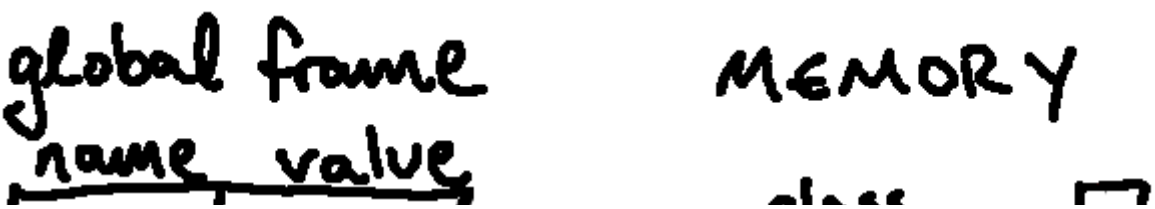
Aqui está a definição da classe:

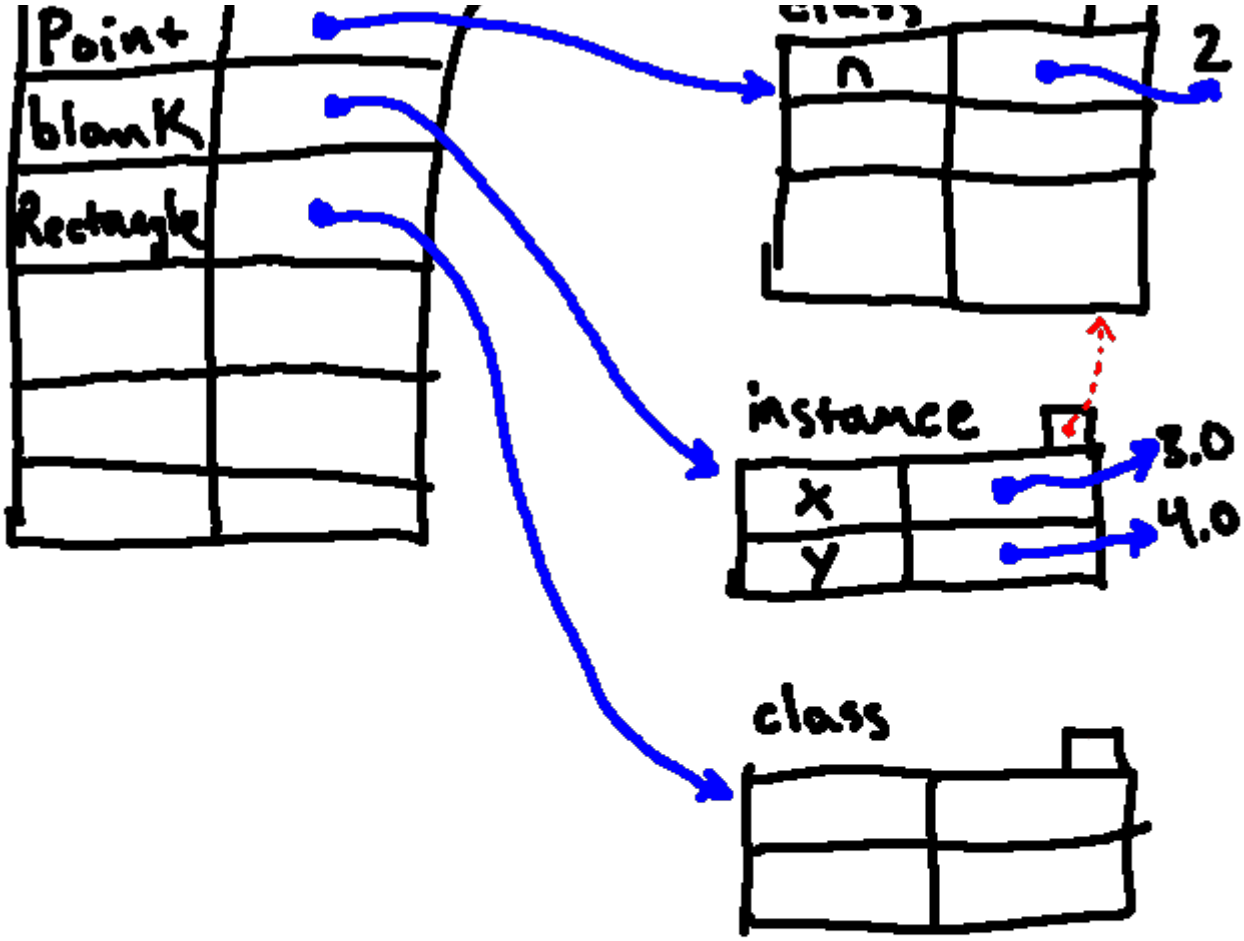
```
class Rectangle:
    pass
```

Não parece muito por agora (porque o corpo não faz nada).

Tente agora:

Desenhe o diagrama do ambiente que resulta da execução desta instrução, supondo que todo o código acima também foi executado.
Mostrar/Esconder





Para representar um retângulo específico, você deve instanciar um objeto Rectangle e atribuir valores aos atributos:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

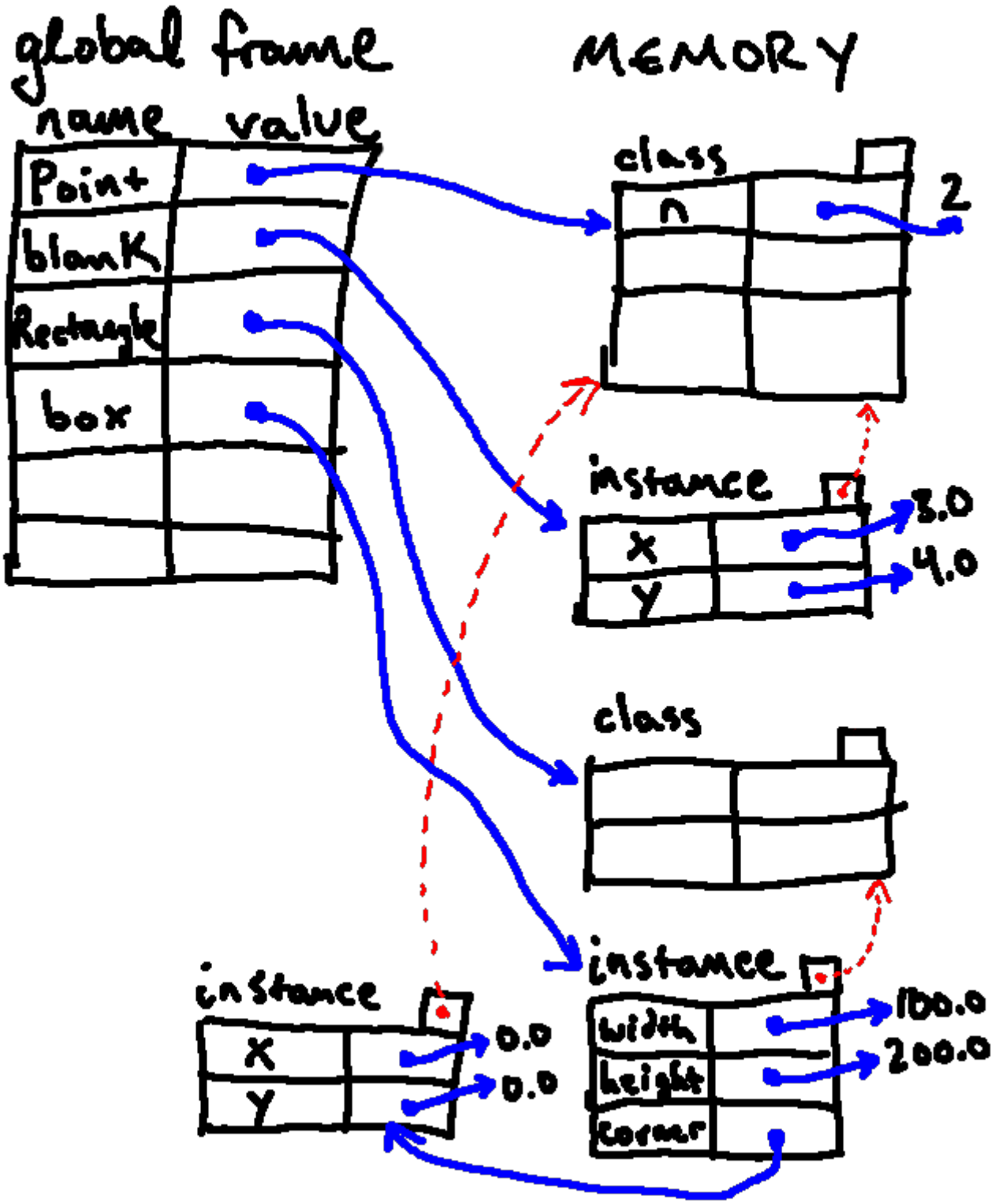
A expressão `box.corner.x` significa: "Vá para o objeto a que `box` se refere e procure o atributo denominado `corner` ; em seguida, vá para esse objeto e procure o atributo denominado `x` ."

Tente agora:

Desenhe o diagrama do ambiente que resulta da execução das instruções acima.
Mostrar/Esconder

Em primeiro lugar, criamos uma instância de `Rectangle` e a associamos com o nome `box` no quadro global. Em seguida, associamos os atributos `width` e `height` dentro dessa instância com os valores `100.0` e `200.0`, respectivamente. Finalmente, fazemos uma nova instância de `Point`, que associamos com o nome `corner` dentro da instância de `Rectangle` e associamos os atributos `x` e `y` *nessa* instância com os valores `0.0` e `0.0`, respectivamente.

Está começando a parecer um pouco com uma tigela de espaguete, mas no final, isso resulta no seguinte diagrama:



Tente agora:

Usando o diagrama de ambiente acima, preveja o que será impresso na tela se executarmos cada um dos seguintes:

```
print(box.width)
print(box.corner.x)
print(box.corner.n)
print(box.corner.box)
print(box.x)
```

Mostrar/Esconder

- O primeiro procurará `box` no ambiente global e, em seguida, procurará o nome `width` dentro desse objeto, encontrando `100.0`.
- O segundo irá procurar `box` no ambiente global, procurar o nome `corner` dentro desse objeto e procurar o nome `x` dentro *desse* objeto, encontrando `0.0`.
- O terceiro pesquisará `box` no ambiente global, pesquisará o nome `corner` dentro desse objeto e pesquisará o nome `n` dentro *desse* objeto. Ele não encontra `n` dentro desse objeto, então ele olha dentro de sua classe (`Point`) e encontra o valor `2`.
- O quarto não imprimirá nada, mas resultará em um erro. Procurando `box.corner` encontra a instância `Point` que criamos. Tentamos pesquisar `box` dentro desse objeto e não encontramos nada, então olhamos dentro da classe. Na classe, novamente não encontramos nada chamado `box`, então desistimos e retornamos um erro. É importante ressaltar que não procuramos por `box` no ambiente global.
- Este último também resultará em erro, pois o nome `x` não existe na instância a que `box` se refere,

3) Classes e Funções

Vimos na seção anterior que instâncias de classes definidas pelo usuário podem ser tratadas como objetos primitivos. Isso significa que também podemos fazer funções que operam em instâncias ou que retornam novas instâncias.

As funções podem retornar instâncias de classes definidas pelo usuário. Por exemplo, `find_center` definido abaixo recebe um `Rectangle` como argumento e retorna um `Point` que contém as coordenadas do centro do retângulo:

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

Aqui está um exemplo que passa `box` como um argumento e atribui o `Point` resultante a `center`:

```
def print_point(p):
    print('(', p.x, ',', p.y, ')')

center = find_center(box)
print_point(center)  # exibe ( 50, 100 )
```

3.1) Funções que Modificam Objetos

Vimos na seção anterior que objetos são mutáveis. Você pode alterar o estado de um objeto fazendo uma atribuição a um de seus atributos. Por exemplo, para alterar o tamanho de um retângulo sem alterar sua posição, você pode modificar os valores de `width e height`:

```
box.width = box.width + 50
box.height = box.height + 100
```

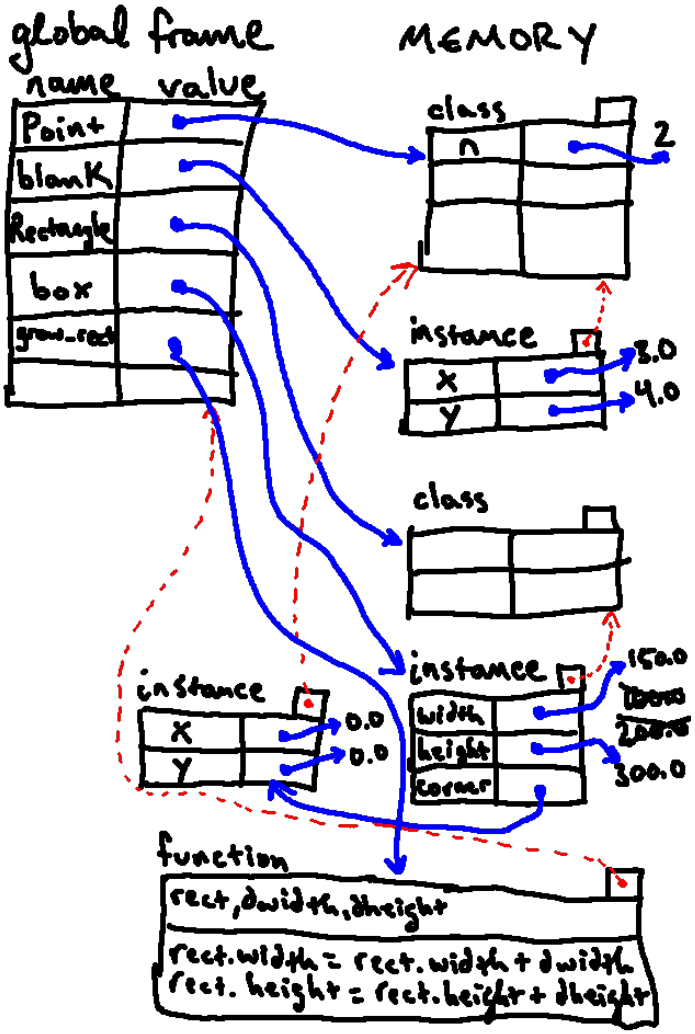
Você também pode escrever funções que modificam objetos. Por exemplo, `grow_rectangle` recebe um objeto Rectangle e dois números, `dwidth e dheight`, e adiciona os números à largura e altura do retângulo:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width = rect.width + dwidth
    rect.height = rect.height + dheight
```

Aqui está um exemplo que demonstra o efeito:

```
print(box.width, box.height)  # exibe 150.0 300.0
grow_rectangle(box, 50, 100)
print(box.width, box.height)  # exibe 200.0 400.0
```

Então, como isso aconteceu? Vai ser um pouco confuso, mas vamos dar uma olhada no diagrama de ambiente:



<< Primeiro Passo | < Passo Anterior | Próximo Passo > | Último Passo >>

PASSO 1

Antes de chamarmos `grow_rectangle` , nós a definimos no quadro global, resultando no diagrama acima.

Tente agora:

Escreva uma função chamada `move_rectangle` que recebe uma instância de `Rectangle` e dois números chamados `dx` e `dy`. Ele deve alterar a localização do retângulo adicionando `dx` à coordenada `x` de `corner` e adicionando `dy` à coordenada `y` de `corner`.
Mostrar/Esconder

Aqui está uma solução:

```
def move_rectangle(rect, dx, dy):
    rect.corner.x = rect.corner.x + dx
    rect.corner.y = rect.corner.y + dy
```

3.2) Funções puras

Também é possível definir *funções puras* envolvendo instâncias (ou seja, funções que não modificam seus argumentos, mas retornam um novo resultado).

Por exemplo, poderíamos escrever uma função como `move_rectangle`, mas que cria uma nova instância que representa o retângulo movido, em vez de alterar sua entrada:

```
def shifted_rectangle(rect, dx, dy):
    new_rect = Rectangle() # cria uma nova instância
    # altura e largura devem ser iguais
    new_rect.width = rect.width
    new_rect.height = rect.height
    # o canto do retângulo é diferente, contudo
    new_rect.corner = Point() # importante, fazemos uma nova instância
de Point para o canto
    new_rect.corner.x = rect.corner.x + dx
    new_rect.corner.y = rect.corner.y + dy
    return new_rect
```

4) Classes e Métodos

Na seção anterior, vimos exemplos de funções que operam em instâncias que criamos. Da mesma forma, poderíamos definir algumas novas funções para definir cálculos relacionados à classe `Point`:

```
import math

def distance_to_origin(pt):
    return (pt.x ** 2 + pt.y ** 2) ** 0.5

def euclidean_distance(pt1, pt2):
    return ((pt1.x - pt2.x) ** 2 + (pt1.y - pt2.y) ** 2) ** 0.5

def manhattan_distance(pt1, pt2):
    return abs(pt1.x - pt2.x) + abs(pt1.y - pt2.y)

def add_vectors(pt1, pt2):
    new_pt = Point()
    new_pt.x = pt1.x + pt2.x
    new_pt.y = pt1.y + pt2.y
    return new_pt

def angle_between(pt1, pt2):
    vert = pt2.y - pt1.y
    horiz = pt2.x - pt1.x
    return math.atan2(vert, horiz)  # calcula arctangente
```

No entanto, apenas olhando para um programa escrito nesse estilo, não é óbvio que haja qualquer conexão entre as funções que definimos e os tipos de dados em que operam. Com alguma observação, entretanto, torna-se aparente que todas as operações acima tomam pelo menos uma instância de `Point` como argumento.

Essa observação é a motivação para *métodos*. Um método é uma função associada a uma classe específica. Os métodos são semanticamente iguais às funções, mas existem duas diferenças sintáticas:

- Os métodos são definidos dentro de uma definição de classe para tornar explícita a relação entre a classe e o método.
- A sintaxe para chamar um método é diferente da sintaxe para chamar uma função. Vamos começar transformando uma dessas funções em um método. Como uma primeira etapa, tudo o que precisamos fazer é mover a definição para a classe (observe a mudança na indentação):

```
class Point:
  n = 2

  def distance_to_origin(pt):
      return (pt.x ** 2 + pt.y ** 2) ** 0.5
```

Uma vez que definimos `distance_to_origin` dessa forma, agora temos duas maneiras de chamá-lo. O primeiro pode parecer familiar: se tivermos uma instância `p` de ponto, podemos procurar a função com `Point.distance_to_origin` e chamá-la com `p` como um argumento:

```
p = Point()
p.x = 3.0
p.y = 4.0
print(Point.distance_to_origin(p))  # exibe 5.0
```

Usando essa notação, Python primeiro procura o nome `Point`. Em seguida, ele procura o nome

`distance_to_origin` dentro desse objeto e chama a função resultante com a instância `p` passada como um argumento.

Acontece que as pessoas raramente chamam métodos usando essa sintaxe. Em vez disso, tendemos a usar uma notação mais concisa para chamar métodos:

```
print(p.distance_to_origin()) # também imprime 5.0
```

Isso pode parecer estranho! Definimos `distance_to_origin` para receber um único argumento, mas acima, parece não levar nenhum! Se isso parece estranho, isso é normal. O que temos aqui é um pouco de algo frequentemente referido como "açúcar sintático", que é um pouco de sintaxe para tornar uma operação comum mais fácil de escrever (assim chamada porque torna a escrita de código um pouco mais doce). Pode ser confuso no início, mas podemos modelar o que o Python está fazendo aqui, então não precisa ser confuso por muito tempo!

Nessa ligeira mudança de notação, `distance_to_origin` ainda é o nome do método que queremos chamar, e seu argumento `pt` ainda é o objeto no qual a função atua. Nos bastidores, porém, se um método for pesquisado em uma *instância* em vez de em uma classe, o Python irá inserir automaticamente essa instância como o primeiro argumento para o método (portanto, neste caso, a instância `p` está associada ao nome `pt` dentro do corpo do método).

Nota

Na verdade, já vimos essa sintaxe antes! Quando usamos `append` para adicionar elementos ao final de uma lista `x`, fizemos isso dizendo `x.append(elt)`. Mas também poderíamos ter feito isso usando `list.append(x, elt)`.

Por convenção, o primeiro parâmetro de um método é chamado `self`, então seria mais comum escrever `distance_to_origin` assim:

```
class Point:
    n = 2

    def distance_to_origin(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5
```

O motivo desta convenção é uma metáfora implícita:

- A sintaxe para uma chamada de função, `distance_to_origin(pt)`, sugere que a função é o agente ativo. Diz algo como, "Ei `distance_to_origin`! Aqui está um ponto para o qual eu quero que você calcule a distância."
- Nessa nova perspectiva (geralmente chamada de *programação orientada a objetos*), os objetos são os agentes ativos. Uma chamada de método como `p.distance_to_origin()` diz "Ei, `p`! Por favor, me diga sua própria distância até a origem." Essa mudança de perspectiva pode ser mais educada, mas não é óbvio que seja útil. Nos exemplos que vimos até agora, pode não ser. Mas, às vezes, transferir a responsabilidade das funções para os objetos torna possível escrever funções (ou métodos) mais versáteis e torna mais fácil manter e reutilizar o código.

4.1) O que é self?

Nesta seção, expandiremos um pouco mais sobre `self` . O texto acima ("O que é `self` ?") pode parecer uma questão filosófica profunda, mas não vamos abordá-la nesse nível. Em vez disso, veremos o que exatamente `self` significa em Python.

Em primeiro lugar, é importante notar que `self` é *apenas um nome*, que normalmente é usado como o primeiro parâmetro de um método. Este primeiro argumento, independentemente de como é chamado, sempre denota *a instância que está sendo operada no momento*.

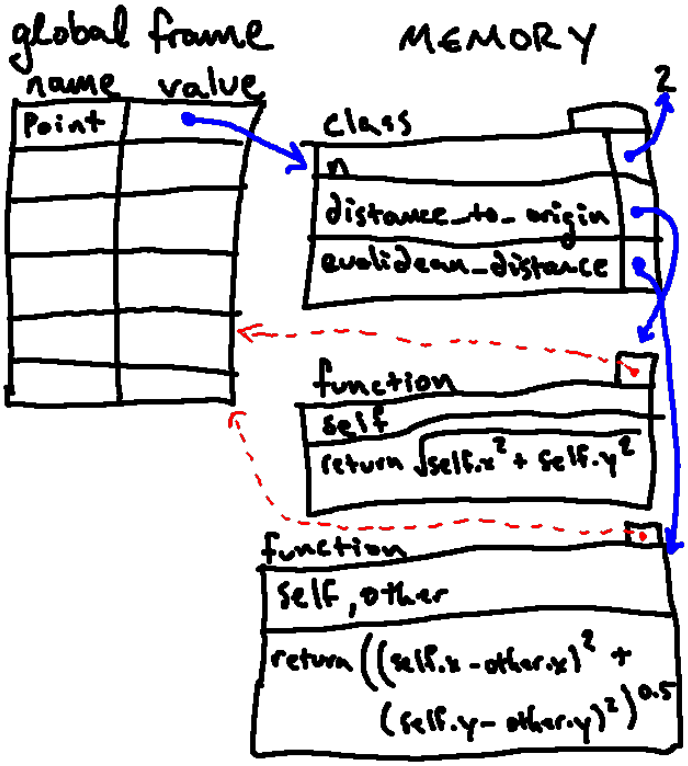
Se um método é pesquisado por meio de uma *classe*, cabe ao programador fornecer a instância dessa classe na qual o método deve atuar. Se o método for pesquisado por meio de uma *instância*, entretanto, o Python inserirá automaticamente essa instância como o primeiro argumento para o método (o argumento normalmente chamado de `self`).

Vamos expandir a definição de `Point` acima para incluir mais um método e, em seguida, invocar alguns métodos:

```
class Point:
    n = 2

    def distance_to_origin(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

    def euclidean_distance(self, other):
        return ((self.x - other.x) ** 2 + (self.y - other.y) ** 2) **
0.5
```



<< Primeiro Passo	< Passo Anterior	Próximo Passo >	Último Passo >>

PASSO 1

Após a definição da classe, temos o diagrama acima. Observe que `distance_to_origin` e `euclidean_distance` dentro do objeto de classe apontam para *funções*. Observe também que os ponteiros-pai dessas funções ainda apontam para o *quadro* no qual foram definidos (aqui, o quadro global), embora eles foram definidos dentro de uma classe.

Tente agora:

Modifique a definição de `Point` para converter as outras funções acima (`angle_between` , `manhattan_distance` , `add_vectors`) em métodos. Como você chamaria esses métodos de uma instância `p` de `Point` ?

4.1.1) Por que self é útil?

Uma razão pela qual `self` é útil é porque ele nos permite acessar e modificar atributos de dentro de um método. Enquanto as variáveis definidas dentro de uma função são acessíveis apenas a partir desse quadro local, os atributos definidos dentro de `self` serão acessíveis a partir de outras chamadas de método e, na verdade, de fora do objeto!

Isso é importante para objetos com valores únicos para uma *instância* específica em chamadas de função (como a classe `Point` com `x` e `y` ; ou a classe `Rectangle` com `corner` , `height` e `width`).

5) Método init

Nos exemplos anteriores, era meio chato criar novas instâncias das classes que definimos; tínhamos que primeiro criar a instância e, em seguida, vincular novas variáveis dentro do objeto resultante.

O método init (abreviação de "initialization") fornece um meio de facilitar esse processo. É um método especial que é chamado quando um objeto é instanciado. Seu nome completo é `__init__` (dois caracteres de underscore, seguidos de `init` e, em seguida, mais dois underscores). Um método init para a classe `Point` pode ser assim:

```
# dentro da classe Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Depois de definir este método, podemos criar uma instância como:

```
mypoint = Point(2, 3)
```

Se os argumentos forem passados durante a criação de uma instância de uma classe, Python os passará para o método `__init__` da classe. Em particular, Python fará o seguinte em resposta ao código acima:

1. Cria uma nova instância de `Point` e vincula-a ao nome `mypoint` .
2. Executa o método `__init__` com `mypoint` passado como o primeiro argumento (no exemplo acima, `Point.__init__(mypoint, 2, 3)`).
3. Dentro do corpo do método, o nome `self` se refere a esta nova instância, então o método armazenará os valores `2` e `3` como `x` e `y` , respectivamente, dentro da instância (para que eles

sejam acessíveis de fora do método como `mypoint.x` e `mypoint.y`). É comum que os parâmetros de `__init__` tenham os mesmos nomes dos atributos. A declaração:

```
self.x = x
```

armazena o valor do parâmetro `x` como um atributo de `self` (a instância recém-criada).

6) Alguns Outros Métodos "Mágicos"

Python também possui vários outros métodos "mágicos", que são chamados automaticamente pelo Python em certas situações e que são tipicamente denotados com um nome cercado por underscores duplos (como init acima). Uma lista mais completa está disponível aqui (https://docs.python.org/3/reference/datamodel.html#special-method-names), mas a seguir estão alguns exemplos:

- `__str__(self)` deve retornar uma string; este método é chamado quando a instância em questão é impressa, ou ao converter para uma string com a função `str`.
- `__add__(self, other)` é chamado quando o operador `+` é usado em duas instâncias, permitindo-nos usar o operador `+` em instâncias de nossas classes personalizadas.
- `__mul__(self, other)` é chamado quando o operador `*` é usado em duas instâncias. Como exemplo, considere imprimir uma das instâncias `Point`. O Python fará o possível para imprimir um resumo útil do objeto, mas o melhor que pode fazer é algo como:

```
p = Point(2, 3)
print(p)  # exibe: <__ main __. Point object at 0x7f8d7fae9d68>
```

Não é muito útil! Mas se definirmos um método `__str__` primeiro, podemos obter resultados muito melhores:

```
# dentro da classe Point:

  def __str__(self):
      return "Point(" + str(self.x) + ", " + str(self.y) + ")"

p = Point(8, 4)
print(p)  # exibe: Point(8, 4)
```

Tente agora:

Anteriormente, quando falávamos sobre diferentes representações para pontos, listamos três possibilidades:

- armazenar `x` e `y` separadamente como duas variáveis,
- armazenar coordenadas em uma lista ou tupla, ou
- criando um novo tipo para representar pontos como objetos. Quais são as vantagens e desvantagens de cada uma dessas representações?

Mostrar/Esconder

Qualquer uma dessas representações poderia, de fato, funcionar bem, mas a representação de um ponto usando uma classe tem uma série de recursos interessantes:

- É fácil criar várias instâncias da classe `Point`, o que não seria verdade se simplesmente

armazenássemos as coordenadas `x` e `y` separadamente.
- A criação da classe `Point` nos permite associar vários métodos específicos para pontos aos próprios objetos, em vez de tê-los como funções autônomas separadas (que precisaríamos se estivéssemos representando pontos como listas ou tuplas).

7) Resumo

Neste conjunto de leituras, aprendemos sobre outro novo recurso poderoso do Python: *classes*, que nos permitiram definir novos *tipos* de objetos Python.

Assim como funções nos permitem abstrair detalhes de *operações* específicas e tratá-los como se tivessem sido integrados ao Python, as classes nos permitem abstrair detalhes de *tipos de dados* específicos e tratá-los como se tivessem sido integrados ao Python.

Neste conjunto de exercícios, você obterá alguma prática na definição de classes personalizadas. No próximo conjunto de leituras e exercícios, aprenderemos sobre alguns outros recursos interessantes de