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

Estamos chegando ao fim de nossa jornada juntos. Nesta seção, não temos muito a apresentar na forma de novo material. Como tal, a tarefa desta semana é amplamente centrada em torno de um exercício de programação maior, com poucos exercícios.

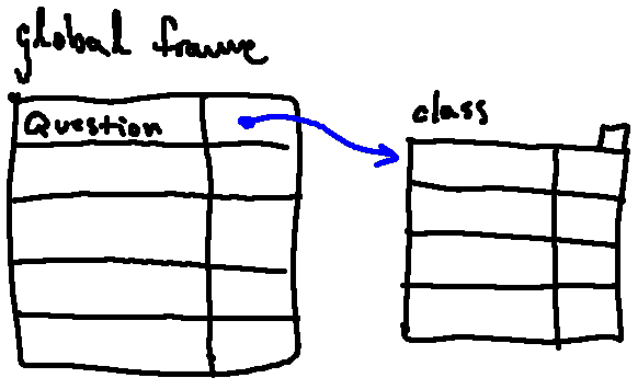
Nesta seção, apresentaremos dois novos tópicos principais, bem como alguns "atalhos" úteis do Python que podem ajudá-lo a escrever um código mais conciso, claro e eficiente.

1) Revisão de Classes por Exemplo

No último conjunto de leituras, introduzimos a ideia de *classes*, o que nos permitiu definir nossos próprios tipos de dados personalizados para uso em nossos programas Python.

Como uma revisão, vamos considerar o seguinte código, que inclui uma classe que pode ser usada em um sistema para aceitar respostas a perguntas e verificar se estão corretas:

```
class Question:
    def __init__(self, p, s):  # nomes ruins de variáveis escolhidos de
propósito para ilustração
        self.prompt = p
```



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

PASSO 1

Como uma primeira etapa ao executar a definição da classe, Python cria um novo objeto de classe e associa o nome `Question` a ele em o quadro global, resultando no diagrama mostrado acima.

Há muitos comportamentos estranhos aqui e pode demorar um pouco para internalizá-los. Em particular, o uso de `__init__` pelo Python e sua passagem implícita do primeiro argumento (geralmente chamado de `self`) podem ser confusos. Portanto, é importante dedicar um tempo para entender essas coisas com cuidado agora. Com o tempo e a prática, você começará a ter mais noção de como essas coisas se comportam.

2) Herança (*Inheritance*)

Às vezes, você pode querer criar uma classe que se comporte de várias maneiras como outro tipo de classe, mas que difira em alguns aspectos. Uma maneira de fazer isso seria copiar toda a definição da classe original e fazer as modificações necessárias. Mas, como muitas outras linguagens com recursos orientados a objetos, o Python nos oferece uma maneira mais fácil de realizar essa mesma tarefa: um mecanismo chamado *herança*.

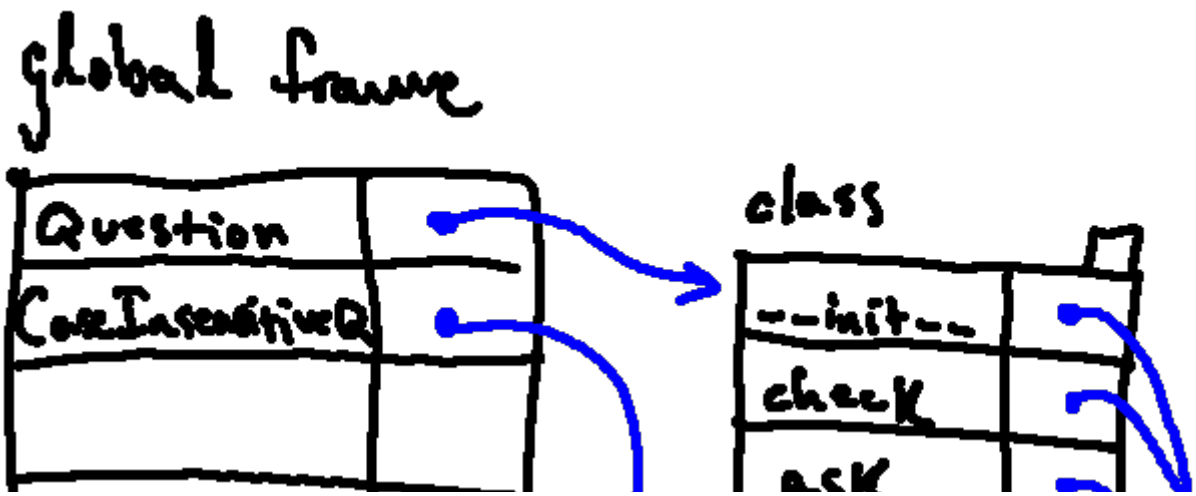
Por exemplo, imagine que queremos criar uma classe que se comporte como a classe `Question` em todos os sentidos, *exceto* que não considera a capitalização da resposta (observe que no exemplo acima, `"blue"` não seria uma resposta aceitável porque `"blue" != "Blue"`). Essa relação (semelhante, mas diferente) é onde herança pode ser interessante.

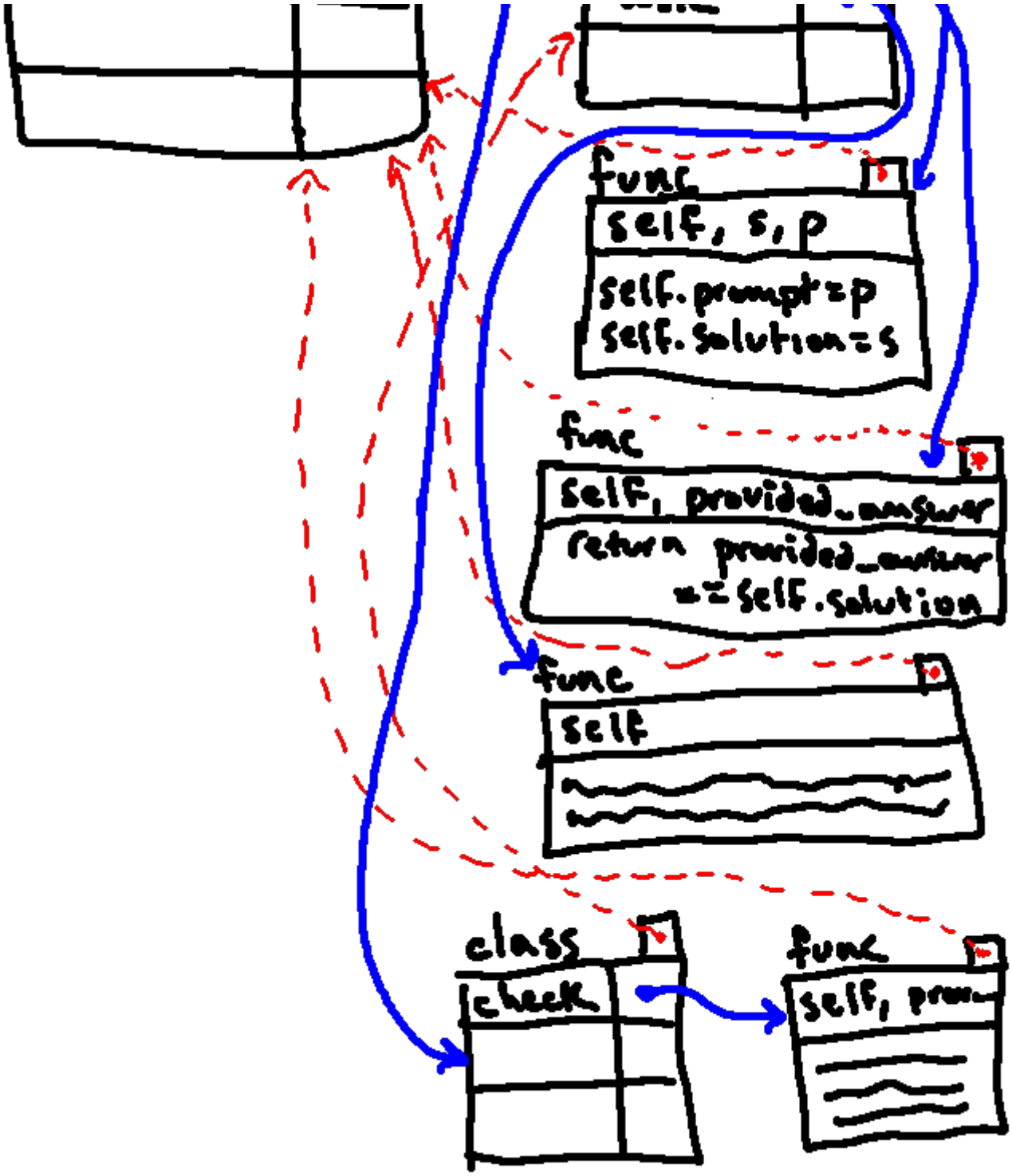
Para definir uma nova classe que herda de uma classe existente, você coloca o nome da classe existente entre parênteses após o nome da nova classe. Por exemplo, considere a seguinte classe:

```
class CaseInsensitiveQuestion(Question):
    def check(self, provided_answer):
        return self.solution.lower() == provided_answer.lower()
```

Observe que a principal diferença visual de outras definições de classe que vimos é o nome `Question` entre parênteses após o nome da classe. Isso indica que `CaseInsensitiveQuestion` herda de `Question`.

Em termos de nossos diagramas de ambiente, o processo de definir essa nova classe se comporta basicamente da mesma maneira que definir qualquer outra classe, mas com uma pequena diferença: essa nova classe terá um ponteiro pai para a classe da qual herda. Por exemplo, se tivermos definido `Question` e `CaseInsensitiveQuestion` apenas em um arquivo, nosso diagrama de ambiente se pareceria com o seguinte:





(mais uma vez alguns dos nomes aqui estão abreviados para fazê-los caber nas caixas e o código para o corpo da nova função `check` também não está escrito para salvar espaço)

O que isso significa em termos práticos? O importante a notar é que, por causa do ponteiro pai, quando procuramos por um nome dentro de `CaseInsensitiveQuestion` e não o encontramos, continuamos procurando na classe `Question` .

Tente agora:

As funções que o Python encontra para cada um dos nomes a seguir foram definidas na classe `Question` ou na classe `CaseInsensitiveQuestion` ?

- `CaseInsensitiveQuestion.__init__`
- `CaseInsensitiveQuestion.check`
- `CaseInsensitiveQuestion.ask`

Mostrar/Esconder

`__init__` e `ask` ainda se referem aos mesmos métodos que foram definidos na classe `Question` original, mas `check` se refere ao novo método definido na classe `CaseInsensitiveQuestion` .

Isso é o que queremos dizer com *herdar* valores. Quando procuramos certos valores da nova classe (referido como "filha" ou "subclasse"), acabamos encontrando exatamente os mesmos objetos que foram definidos na classe antiga (o "pai" ou "superclasse") . Nesse sentido, diríamos que `CaseInsensitiveQuestion` *herda de* `Question` .

Observe, porém, que esse mecanismo ainda nos permitiu sobrescrever os valores que queríamos alterar da classe original (já que o Python primeiro procurará nomes na classe filha).

Tente agora:

Considere o seguinte trecho de código:

```
q2 = CaseInsensitiveQuestion("What is your favorite animal?", "CAT")
q2.ask()
```

O que acontecerá quando este código for executado se o usuário digitar `cat` no prompt? Para cada método, estamos executando a definição da classe `Question` ou da classe `CaseInsensitiveQuestion` ?

Mostrar/Esconder

Este código fará o seguinte:

`q2 = CaseInsensitiveQuestion("What is your favorite animal?", "CAT")`

- cria uma nova instância de `CaseInsensitiveQuestion` e associa-a ao nome `q2` no quadro global. Observe que esta instância terá um ponteiro pai para a classe `CaseInsensitiveQuestion` , que por sua vez tem um ponteiro para a classe `Question` .
- encontra o método `__init__` para chamar olhando primeiro em `CaseInsensitiveQuestion` e, não encontrando nada lá, procurando em `Question` .
- chama esse método com nossa nova instância passada como o primeiro argumento (`self`) e as duas strings como o segundo e o terceiro argumentos, respectivamente (observe que isso armazenará esses valores como `prompt e solution` dentro dessa nova instância).

`q2.ask()`

- chama o método `ask` (encontrado olhando em `q2` , então em `CaseInsensitiveQuestion` , então em `Question` onde ele é finalmente encontrado) com esta instância (chamada `q2` no quadro global) passada como o primeiro argumento (`self`). No processo de chamar esta função:
  - imprime o valor de `self.prompt` na tela. Como neste quadro `self` está vinculado à nossa instância (aquela conhecida como `q2` no quadro global), procuramos a variável `prompt` nessa instância e encontramos a string `"What is your favorite animal?"` .
  - pede a entrada do usuário. Se o usuário digitar `cat` no prompt, isso significa que o nome

`response` (no quadro local, *não* na instância) irá se referir à string `"cat"` .

- executa `self.check(response)` . Como `self` está vinculado à nossa instância neste quadro, o Python procurará o método de verificação nessa instância e, em seguida, em `CaseInsensitiveQuestion` , onde encontrará o novo método. Em seguida, ele chama esse método com essa instância (a chamada `q2` no quadro global) passada implicitamente como o primeiro argumento (`self`) e a string `"cat"` como o segundo argumento. No processo de chamar esta função:
  - compara `self.solution.lower()` com `provided_answer` . Nesse caso, como `self` está vinculado à nossa instância no quadro local, `self.solution` é a string `"CAT"` . `response` é a string `"cat"` . Como as versões em minúsculas dessas strings são realmente idênticas, o método `check` retornará `True` .
- Como a chamada para `self.check(response)` retornou `True` , a chamada `self.ask()` imprimirá `"Yay! That's correct"` e sair, retornando `None` porque nenhum valor de retorno foi especificado.

Como sua resposta ao acima seria diferente se `q2` fosse uma instância de `Question` em vez de `CaseInsensitiveQuestion` ?

Mostrar/Esconder

O processo de execução do código acima teria ocorrido basicamente conforme descrito acima, exceto que o método `check` chamado teria sido o de `Question` em vez de `CaseInsensitiveQuestion` . Dessa forma, essa chamada de método teria retornado `False` , o que resultaria em `"Incorrect"` sendo impresso na tela.

3) Other Goodies

Ironicamente, um dos objetivos nessas leituras foi ensinar a você o mínimo possível de Python. A ideia foi concentrar na modelagem precisa de um pequeno número de recursos do Python, para focar em estruturas centrais *que não são exclusivas do Python*, e fornecer uma base sobre a qual futuras peças podem ser adicionadas com relativa facilidade. Quando havia duas maneiras de fazer algo, uma foi introduzida e a outra não foi mencionada. Ou às vezes a segunda foi colocada em um exercício.

Agora vamos voltar para algumas das coisas boas que ficaram para trás. O Python oferece *muitos* recursos que não são realmente necessários (você pode escrever bom código para resolver qualquer problema sem eles), mas com eles você pode, às vezes, escrever um código mais conciso, legível ou eficiente e, às vezes, todos os três.

Nesta uma seção, não há tempo para cobrir tudo, mas você terá muito tempo enquanto continua com a programação para pegar mais e mais dessas pequenas peças!

3.1) Slicing

Vimos nas seções anteriores que era possível "indexar em" sequências (strings, listas, tuplas) para extrair determinados valores delas. Também é possível selecionar um segmento (chamado de *fatia* ou *slice*) da sequência usando uma sintaxe semelhante.

Considere o seguinte exemplo usando strings:

```
fruit = 'banana'
print(fruit[0: 4])  # imprime bana
print(fruit[2: 4])  # imprime na
```

O operador de *slicing* `[start:stop]` retorna a parte da string desde o $start$-ésimo caractere até o $stop$-ésimo caractere, incluindo o primeiro, mas excluindo o último.

Se você omitir o primeiro índice (antes dos dois pontos), a fatia começa no início da string. Se você omitir o segundo índice, a fatia vai até o final da string:

```
print(fruit[:3])  # imprime ban
print(fruit[3:])  # imprime ana
```

Se o primeiro índice for maior ou igual ao segundo, o resultado será uma string vazia, representada por duas aspas:

```
print(fruit[3:3])  # isso resulta na string vazia ""
```

Uma string vazia não contém caracteres e tem comprimento 0, mas além disso, é igual a qualquer outra string.

Continuando com este exemplo, o que você acha que `fruit[:]` significa? Experimente e veja.

O operador de fatiamento (de forma semelhante à função `range`) também aceita um terceiro argumento opcional (comumente referido como `step`), que especifica quantos caracteres vamos 'pular' em cada etapa. Se não for especificado, o valor padrão é `1`.

Em sua forma completa, esse operador se parece com `[start:stop:step]`. Por exemplo:

```
print(fruit[::2])  # bnn
print(fruit[0:4:3])  # ba
print(fruit[1::3])  # an
```

Esses números às vezes podem ser um pouco confusos; mas para mim, ajuda pensar sobre esse operador em termos de como ele seria definido como uma função:

```
def slice(string, start, stop, step):
    out = ''
    index = start
    while index < stop:
        out += string[index]
        index += step
    return out
```

Observe que `step` pode ser negativo, o que nos dá uma maneira fácil de fazer uma versão *reversa* de uma determinada sequência:

```
print(fruit[::-1])
```

O operador slice também funciona em listas:

```
t = ['a', 'b', 'c', 'd', 'e', 'f']
print(t[1:3])  # ['b', 'c']
print(t[:4])  # ['a', 'b', 'c', 'd']
print(t[3:])  # ['d', 'e', 'f']
```

Se você omitir o primeiro índice, a fatia começa no início. Se você omitir a segunda, a fatia vai para o fim. Portanto, se você omitir ambos, a fatia será uma cópia de toda a lista.

```
print(t[:])  # ['a', 'b', 'c', 'd', 'e', 'f'] (mas esta é uma lista DIF
ERENTE contendo os mesmos elementos)
```

Como as listas são mutáveis, às vezes pode ser útil fazer uma cópia antes de executar operações que modificam as listas.

Um operador de fatia no lado esquerdo de uma atribuição pode atualizar vários elementos:

```
t = ['a', 'b', 'c', 'd', 'e', 'f']
t[1: 3] = ['x', 'y']
print(t)  # ['a', 'x', 'y', 'd', 'e', 'f']
```

3.2) Argumentos Opcionais para Funções

```
def approximately_equal(x, y, threshold=0.1):
    return abs(x - y) <= thresold
```

Se `threshold` for especificado (por exemplo, `aproximately_equal(x, y, 1e-6)`), então esse valor será usado para `threshold` (neste caso, 10^{-6}). Se não (por exemplo, `approximately_equal(x, y)`), então o valor padrão (`0.1`) será usado para `threshold`.

Em outras palavras, se fornecermos o argumento opcional, ele *substituirá* o valor padrão.

Se uma função tiver parâmetros obrigatórios e opcionais, todos os parâmetros obrigatórios devem vir

primeiro, seguidos pelos opcionais.

Há uma pequena advertência aqui: os valores padrão são avaliados no momento em que a função é *definida*, não quando a função é chamada. Portanto, devemos ter cuidado ao usar um objeto mutável como um valor padrão para um parâmetro; se fizermos isso, todas as chamadas para a função usarão o mesmo objeto (que pode ter sido modificado por chamadas anteriores!).

3.3) Expressões Condicionais

Vimos declarações condicionais em um de nossos primeiros conjuntos de leituras. As instruções condicionais são freqüentemente usadas para escolher um de dois valores; por exemplo:

```
def absolute_value(x):
    if x >= 0:
        return x
    else:
        return -x
```

Esta instrução verifica se `x` é positivo e retorna um valor diferente com base no resultado dessa verificação.

Podemos escrever esta declaração de forma mais concisa usando uma *expressão condicional*:

```
def absolute_value(x):
    return x if x >= 0 else -x
```

Você pode quase ler esta linha como em inglês: "esta função retorna `x` se `x` for não negativo; caso contrário, retorna `-x`."

As funções recursivas às vezes podem ser reescritas usando expressões condicionais. Por exemplo, aqui está uma versão recursiva de fatorial:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Podemos reescrever assim:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

3.4) Compreensões de Listas

Nas seções anteriores, vimos um padrão comum relacionado à criação de uma lista de elementos com base em outra lista de elementos. Por exemplo, esta função pega uma lista de strings, mapeia o método string `upper` para os elementos e retorna uma nova lista de strings (maiúsculas):

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

Podemos escrever isso de forma mais concisa usando uma *compreensão de lista*:

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

Os colchetes indicam que estamos construindo uma nova lista. A expressão dentro dos colchetes especifica os elementos da lista, e a cláusula `for` indica a sequência que estamos percorrendo.

A sintaxe de uma compreensão de lista é um pouco estranha porque a variável de loop, `s` neste exemplo, aparece na expressão antes de chegarmos à definição.

As compreensões de lista também podem ser usadas para filtragem.

Por exemplo, se tivéssemos uma lista e quiséssemos encontrar os quadrados dos números ímpares, poderíamos fazer:

```
def squared_odd_numbers(input_list):
    out = []
    for i in input_list:
        if i % 2 == 1:  # if i for ímpar
            out.append(i ** 2)
    return out
```

ou, usando uma compreensão de lista:

```
def squared_odd_numbers(input_list):
    return [i ** 2 for i in input_list if i % 2 == 1]
```

Esta função seleciona apenas os elementos de `input_list` que são ímpares, os eleva ao quadrado e retorna uma nova lista contendo o resultado.

Compreensões de lista são um dos meus recursos favoritos do Python. Eu as uso o tempo todo no código que escrevo porque são concisas e fáceis de ler, pelo menos para expressões simples. E geralmente são mais rápidas do que os loops `for` equivalentes (às vezes muito mais rápidos). Então, se você está com raiva de mim por não mencioná-las antes, eu entendo.

Mas, em minha defesa, as compreensões de lista podem ser mais difíceis de debug porque você não pode colocar uma instrução `print` dentro do loop, e elas não podem fazer tudo que um loop `for` pode fazer (pelo menos não sem algum esforço sério). Elas são divertidos de se experimentar, mas sugiro que você as use somente se o cálculo for simples o suficiente para que você acerte na primeira vez (ou esteja disposto a gastar tempo debugging!).

3.5) Manejando Erros

Em algum ponto ao trabalhar nos exercícios desta aula, você pode ter encontrado erros que eram difíceis de prever. Nesses casos, seria bom ter uma maneira do Python detectar que ocorreu um erro (na linguagem do Python, uma *exceção* porque não é um comportamento normal) e se comportar de maneira adequada.

Em alguns casos, em vez de tentar *prever* erros, é melhor prosseguir e tentar (e lidar com os problemas se eles acontecerem), que é exatamente o que a instrução `try` faz. A sintaxe é semelhante a uma instrução `if` ... `else` :

```
def safe_divide(x, y):
    try:
        return x / y # isso pode resultar em um erro de 'divisão por ze
ro'
    except:
        print("Erro na divisão!")
        return None
```

Python começa executando o corpo `try`. Se tudo correr bem, ele ignora a cláusula `except` e prossegue. Se ocorrer uma exceção, ele sai da cláusula `try` e executa a cláusula `except`.

O tratamento de uma exceção com uma instrução `try` é chamado de "captura" de uma exceção. Neste exemplo, a cláusula `except` imprime uma mensagem de erro que não é muito útil. Em geral, capturar uma exceção dá a você a chance de corrigir o problema, ou tentar novamente, ou pelo menos encerrar o programa normalmente (em vez de um texto vermelho estranho relacionado ao Python).

3.6) "any" e "all"

Python fornece uma função integrada, `any`, que pega uma sequência de valores booleanos e retorna `True` se qualquer um dos valores for `True`. Funciona em listas, por exemplo:

```
print(any([False, False, True]))  # True
```

Mas é frequentemente usado com expressões geradoras:

```
print(any(letter == "t" for letter in "monty"))  # True
```

Esse exemplo não é muito útil porque faz a mesma coisa que o operador `in`. Mas poderíamos usar qualquer um para reescrever comportamentos mais complicados. Por exemplo, considere a seguinte função:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

Poderíamos ter reescrito esta função usando `any`:

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

A função quase se parece com linguagem normal: "`word` evita `forbidden` se não houver nenhuma letra proibida em `word`."

Usar `any` um com uma expressão geradora é eficiente porque ele para imediatamente se encontrar um valor `True`, então não precisa avaliar toda a sequência.

Python fornece outra função integrada, `all`, que retorna `True` se cada elemento da sequência for `True`.

3.7) Unpacking de Sequência

Muitas vezes é útil trocar os valores de duas variáveis. Com atribuições convencionais, você deve usar uma variável temporária. Por exemplo, para trocar `a` e `b`:

```
temp = a
a = b
b = temp
```

Esta solução é incômoda; *atribuição de tupla* é mais elegante:

```
a, b = b, a
```

O lado esquerdo é uma tupla de variáveis; o lado direito é uma tupla de expressões. Cada valor é atribuído à sua respectiva variável. Todas as expressões do lado direito são avaliadas antes de qualquer uma das atribuições.

O número de variáveis à esquerda e o número de valores à direita devem ser iguais:

```
a, b = 1, 2, 3  # isso produz: ValueError: muitos valores para desempac
otar
```

Essa mensagem de erro pode parecer estranha, mas essa ideia de atribuir cada elemento em uma sequência a um nome de variável diferente costuma ser chamada de *desempacotamento* dessa sequência.

Por exemplo, considere as três partes de código a seguir para calcular a distância entre os pontos (onde os pontos são representados como tuplas `(x, y)`).

```
def distance(pt1, pt2):
    return ((pt1[0] - pt2[0]) ** 2 + (pt1[1] - pt2[1]) ** 2) ** 0.5

def distance(pt1, pt2):
    x1 = pt1[0]
    y1 = pt1[1]
    x2 = pt2[0]
    y2 = pt2[1]
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5

def distance(pt1, pt2):
    x1, y1 = pt1
    x2, y2 = pt2
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5
```

A definição da primeira função é boa e direta, mas é um pouco difícil de ler porque as operações de indexação são todas feitas junto com a operação matemática para calcular a distância. A segunda torna a última linha mais fácil de ler, mas à custa de ter que escrever 4 linhas extras de código. A última atinge um bom equilíbrio entre os dois.

Mais geralmente, o lado direito pode ser qualquer tipo de sequência (string, lista ou tupla). Por exemplo, para dividir um endereço de e-mail em nome de usuário e nome de domínio, poderíamos fazer:

```
address = 'a_clever_username@mit.edu'
uname, domain = address.split('@')
```

O valor de retorno de `split` é uma lista com dois elementos; o primeiro elemento é atribuído a

`uname` , o segundo a `domain` .

```
print(uname)  # imprime a_clever_username
print(domain)  # imprime mit.edu
```

3.8) Comparações Múltiplas em Linha

Anteriormente, falamos sobre os operadores de comparação do Python (`>` , `<` , `>=` , `<=` , `==` , `!=`) como se fossem operadores binários, mas na verdade são n-ários (ou seja, eles podem assumir mais de 2 operandos).

Por exemplo, poderíamos escrever:

```
x < y < z
```

que será avaliado como `True` apenas se `x` for menor que `y` *e* `y` for menor que `z` . Isso também funciona para os outros comparadores (e até mesmo para combinações de comparadores diferentes).

Por exemplo: