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

No último conjunto de leituras, apresentamos vários tipos de objetos Python, bem como modelos de como Python avalia expressões e como gerencia o armazenamento e a referência de variáveis. Também introduzimos nosso primeiro meio de controlar a ordem de avaliação das instruções em um programa por meio da execução condicional.

Nesta leitura, apresentaremos e exploraremos vários novos tipos de objetos Python e veremos como encaixar esses novos tipos em nossa estrutura existente. Também apresentaremos alguns novos mecanismos de fluxo de controle muito poderosos.

Antes de mergulharmos no novo material desta tarefa, você pode revisar alguns dos problemas do último conjunto de exercícios. Se tiver problemas com qualquer um deles, você pode querer voltar e revisar as leituras das seções relevantes da tarefa anterior, já que quase tudo apresentado neste conjunto de leituras será baseado nas ideias do último conjunto.

1) *Strings*

No último conjunto de leituras, vimos que poderíamos exibir caracteres na tela literalmente, colocando-os entre aspas em uma declaração `print` . Por exemplo, o seguinte exibirá `hello, python!` na tela:

```
print("olá, python!")
```

Mas, na época, não falamos muito sobre o que essa declaração realmente significava em termos de nosso modelo mental de Python. Nesta seção, começaremos a esclarecer isso um pouco, introduzindo um novo tipo em nosso modelo mental: *strings*.

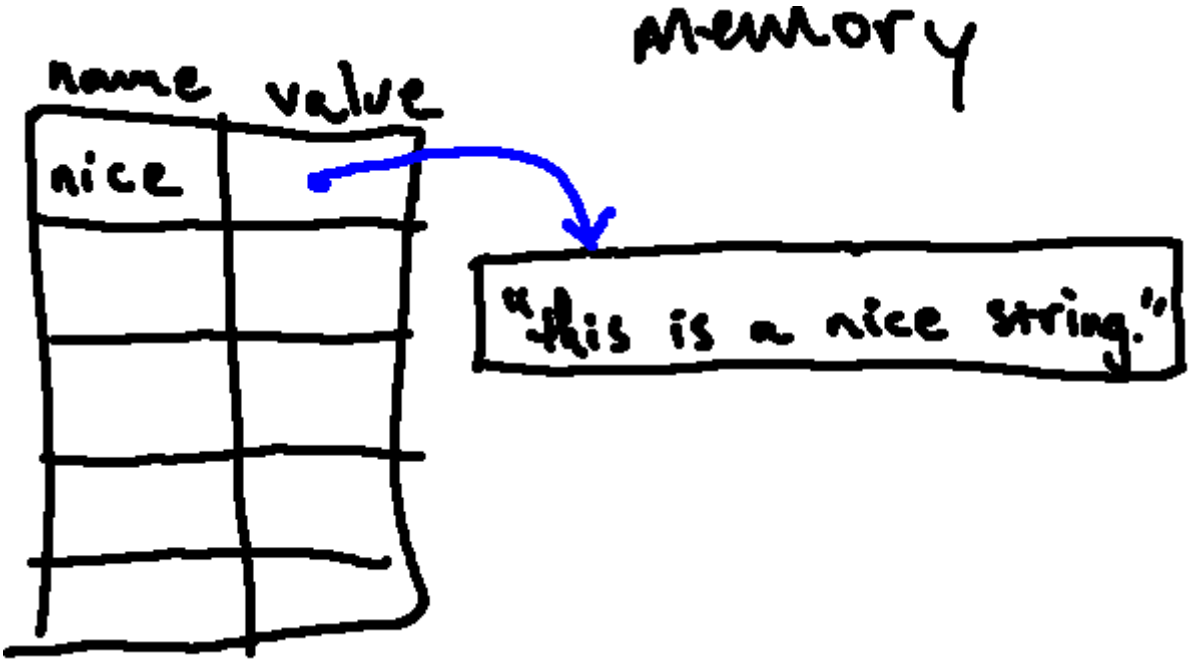
Uma *string* é um tipo que representa uma sequência de caracteres. Em Python, esse tipo recebe o nome de `str` . Podemos usar tanto aspas duplas (`"`) ou aspas simples (`'`) para incluir strings. Portanto, a expressão `"abcd"` é avaliada como uma string, mas `'efgh'` também.

Como as strings são na verdade um tipo de objeto, podemos fazer muito mais do que apenas exibi-las! Podemos, por exemplo, armazenar uma string em uma variável:

```
nice = "This is a nice string."
```

Podemos pensar nisso da mesma forma que pensamos em outras atribuições de variáveis:

- Python vai começar avaliando o valor no lado direito do símbolo `=` (que, neste caso, resulta em uma string).
- Em seguida, armazenará essa string na memória e associará o nome `bela` a ela. Assim como fizemos com objetos `int` e `float`, podemos denotar strings em nossos diagramas de ambiente simplesmente escrevendo seus valores, embora também possa ser uma boa ideia desenhar uma caixa ao redor de uma string para que fique claro que se trata de um único objeto. Executando o pedaço de código acima, por exemplo, resultaria no seguinte diagrama de ambiente:



Assim que tivermos a string armazenada em uma variável, podemos incluir a variável em outras expressões. Por exemplo, depois de fazer a definição acima, poderíamos exibir essa string com:

```
print(nice)
```

Isso irá procurar o nome da variável `nice` no quadro global; fazendo isso, ele encontra a string que está armazenada na memória e a exibe.

Tente agora:

Considere os dois pequenos programas a seguir.

O primeiro programa lê:

```
animal_favorito = "gato"
linguagem_favorita = "python"
print(animal_favorito)
print(linguagem_favorita)
```

e o segundo diz:

```
animal_favorito = "gato"
linguagem_favorita = "python"
print("animal_favorito")
print("linguagem_favorita")
```

Dê uma olhada bem de perto nesses programas. Sintaticamente, qual é a diferença entre esses dois programas? Como essa mudança afeta o significado do programa?

Preveja o que cada programa exibirá. Em seguida, digite cada um no Python e execute-os. Os resultados correspondem às suas previsões?

▶ Mostrar/Esconder

1.1) Operacões em *Strings*

Vimos no último conjunto de leituras que o *tipo* de um objeto é importante para determinar os tipos de operações que podemos realizar naquele objeto. Por exemplo, poderíamos realizar operações aritméticas com objetos `int` e `float`, mas não com objetos `NoneType`. Da mesma forma, podemos realizar alguns tipos de operações em strings. Voltaremos a essa ideia mais tarde, mas, por enquanto, vamos explorar o significado do operador `+` no que se refere a strings.

Tente agora:

Tente executar o seguinte em Python:

```
print("Estou adicionando esta string" + "a esta string")
```

Qual valor é exibido? Tente adicionar mais algumas strings para descobrir exatamente o que o operador `+` faz quando seus operandos são strings.

▶ Mostrar/Esconder

Tente agora:

Desenhe um diagrama de ambiente que mostre o resultado da execução do programa curto a seguir e preveja o que ele exibirá:

```
x = "snow"
y = "ball"

z = x + y

x = "basket" + y

# a string do meio contém um único caractere de "espaço"
print (z + " " + x)
```

▶ Mostrar/Esconder

Tente agora:

Agora vamos examinar o efeito dos tipos de operando em algumas expressões. Em geral, Python pode usar os operadores que discutimos para combinar ou comparar *números* entre si (combinar e comparar objetos `int` com objetos `float` tende a funcionar como esperávamos), mas é importante notar que strings geralmente não podem ser combinadas com números.

Tente prever se as seguintes expressões serão avaliadas sem erros e, em caso afirmativo, tente prever o valor e o tipo que resulta da avaliação de cada uma. Em seguida, digite-os no Python para verificar você mesmo. Se o Python gerar uma mensagem de erro para qualquer um deles, leia com atenção e tente

descobrir o que isso significa e por que aconteceu.

- `6 + 6.0`
- `"6" + "6.0"`
- `6 + "6.0"`
- `6 + "6"`
- `6 == 6.0`
- `"6" == 6`
- `"6" == "6.0"`
- `6.0 == "6.0"`

▶ Mostrar/Esconder

Tente agora:

Agora, vamos examinar o operador `*`. Tente executar o seguinte código em Python:

```
print("cat" * 5)
```

Qual é o resultado? O que o operador `*` faz quando o primeiro argumento é uma string e o segundo é um inteiro?

▶ Mostrar/Esconder

O que acontecerá se o segundo argumento não for um `int`, mas um `float`? Um `NoneType`? Uma `str`? O que acontece se você alterar a ordem dos operandos (ou seja, `5 * "cat"`)?

▶ Mostrar/Esconder

1.2) Convertendo entre os Tipos

No último conjunto de leituras, vimos que poderíamos converter entre objetos `int` e `float` (por exemplo, com `int(7.8)` ou `float(6)`).

Também é possível converter entre strings e tipos numéricos, desde que estejamos lidando com strings em uma forma particular. Por exemplo:

- `str(6.0)` nos dará a string `"6.0"`.
- `int("2")` nos dará o inteiro `2`.
- `float("7.8")` nos dará o float `7.8`.

Tente agora:

O que acontece se você tentar converter outros valores em inteiros e `float`s? Experimente, por exemplo:

- `int("tomate")`
- `int("7.8")`
- `float("6")`

▶ Mostrar/Esconder

2) Strings são Sequências

Strings são um exemplo de um *tipo composto*: são uma sequência de caracteres. As sequências em Python têm várias operações interessantes associadas a elas. Começaremos explorando isso no contexto de strings e, em seguida, generalizaremos para outros tipos de sequências.

2.1) Extraindo Elementos Contidos

Você pode pedir ao Python um caractere de uma string com o operador colchete `[]`. Por exemplo, tente o seguinte:

```
fruta = "banana"
letra = fruta[1]
print(letra)
```

A segunda instrução seleciona o caractere número 1 de `fruta`, armazena-o na memória e associa o nome `letra` a ele. A expressão entre colchetes (neste caso, `1`) é chamada de *índice*. O índice, que deve ser um número inteiro, indica qual caractere na sequência você deseja.

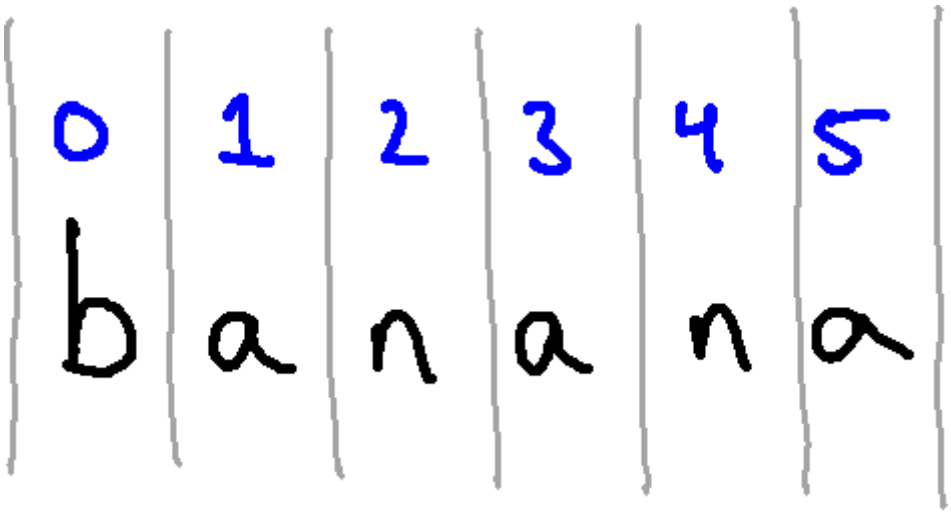
Mas, executando o código acima, você pode não obter a resposta que espera!

Tente agora:

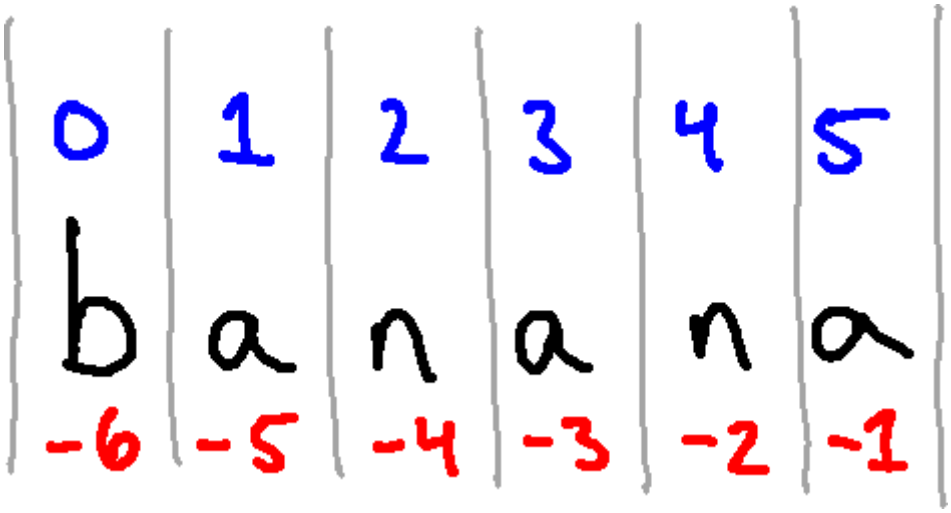
Execute o código acima em Python, observe o resultado (que talvez seja surpreendente!) e continue lendo.

A maioria das pessoas esperaria que o caractere 1 de `"banana"` fosse `"b"`. Mas em Python (como em muitas linguagens de programação), na verdade começamos a contagem em 0 em vez de em 1 (você deve ter notado que nossos números contam a partir de 0 nesta classe também!). Um jeito de pensar nisso é que um índice indica quantos elementos aparecem antes na sequência.

Portanto, os índices de 0 a 5 estão associados às letras da string, conforme mostrado abaixo:



Talvez é importante notar que você também pode indexar a partir do final de uma string. O índice `-1` está associado ao *último* caractere em uma string, `-2` ao penúltimo e assim por diante. Portanto, realmente temos dois índices associados a cada personagem:



Tentar acessar um índice diferente de um desses números (neste caso, inteiros entre `-6` e `5`, inclusos) resulta em erro.

Tente agora:

Tente prever se cada uma das seguintes expressões será avaliada sem erros e, em caso afirmativo, tente prever o valor de cada uma. Depois de fazer suas suposições, exiba-as em Python para verificar. Se o Python gerar uma mensagem de erro para qualquer um deles, leia com atenção e tente descobrir o que isso significa e por que aconteceu.

- `"cat"[0]`
- `"ferret"[5]`
- `"cow"[1] == 'horse'[-4]`
- `int("60.0"[-4])`
- `'hamster'[7]`
- `"tomato"[-4]`

▶ Mostrar/Esconder

2.2) Iteração

Muitos cálculos interessantes em strings envolvem processá-las um caractere por vez. Frequentemente, eles começam no início, selecionam cada caractere por vez, fazem algo com ele e continuam até o fim. Este padrão de processamento de uma string pode ser referido como *looping over* (iterando) a string. Podemos escrever um percurso com uma nova construção Python: um loop `for`. Para começar, vamos considerar o seguinte trecho de código:

```
word = "cat"
for letter in word:
    print(letter)
print('feito')
```

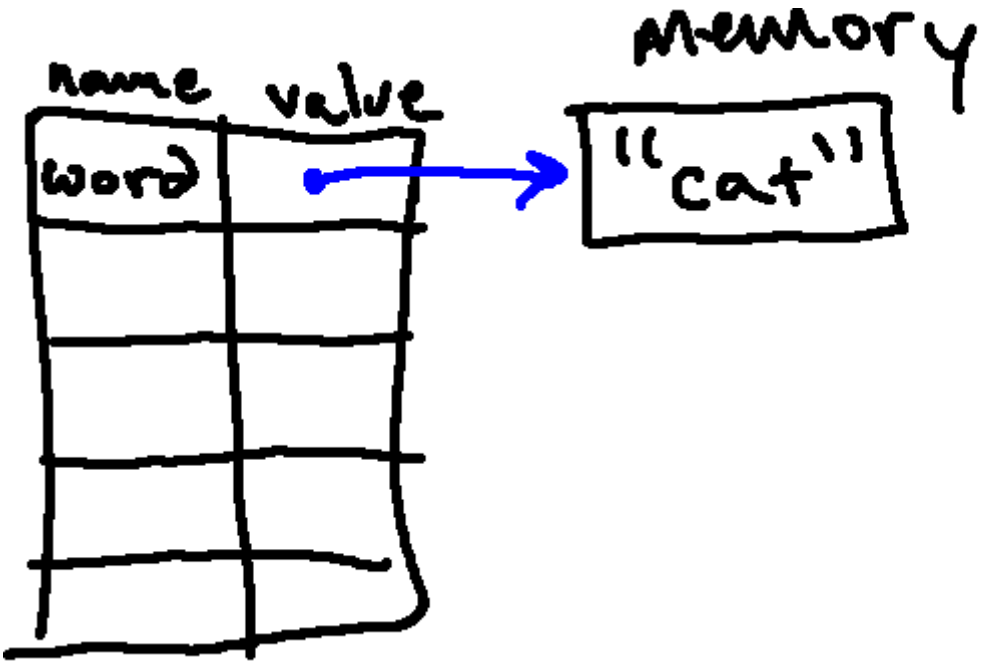
De certa forma, podemos ler isso: para cada letra da string `"cat"`, mostre essa letra (e, depois disso, imprima a palavra `"feito"`). E é exatamente isso que o Python fará. A execução deste código produzirá o seguinte resultado:

```
c
a
t
feito
```

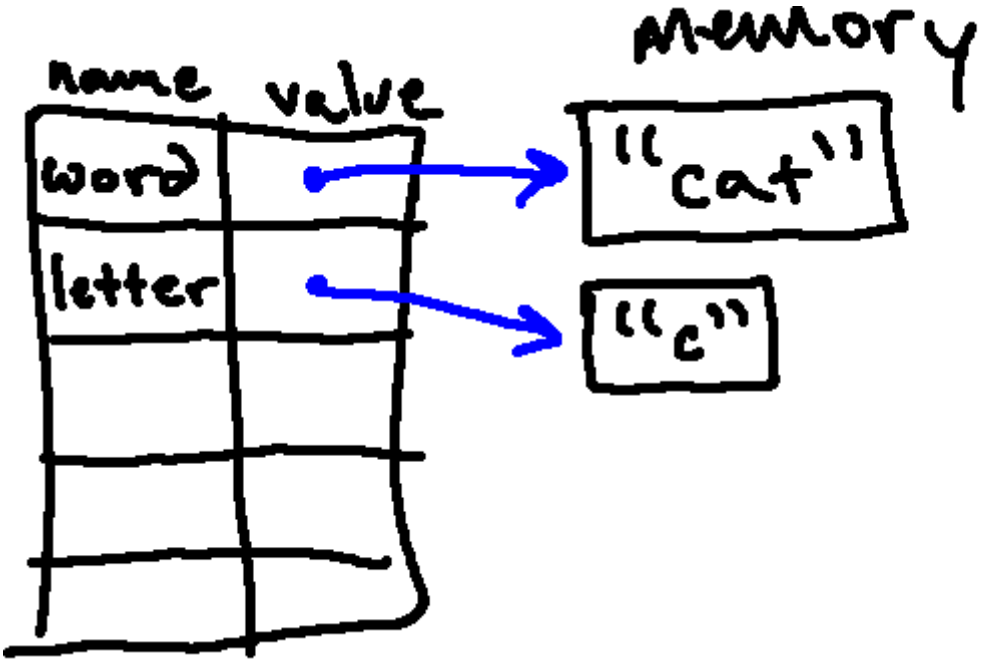
Repetiremos o código na parte indentada (o *corpo*) uma vez para cada elemento em `word`. Antes de cada repetição, o próximo caractere na string é atribuído à variável `letter`. Depois de passar por todos os caracteres, Python continuará com o resto do programa.

De certa forma, essa é uma estrutura mais complicada do que algumas das outras que examinamos, portanto, vamos examinar esse código com muito cuidado e acompanhar como o diagrama de ambiente muda com o tempo.

Depois de executar a primeira linha (`word = "cat"`), temos a string `"cat"` na memória, e ela está associada ao nome `word`:



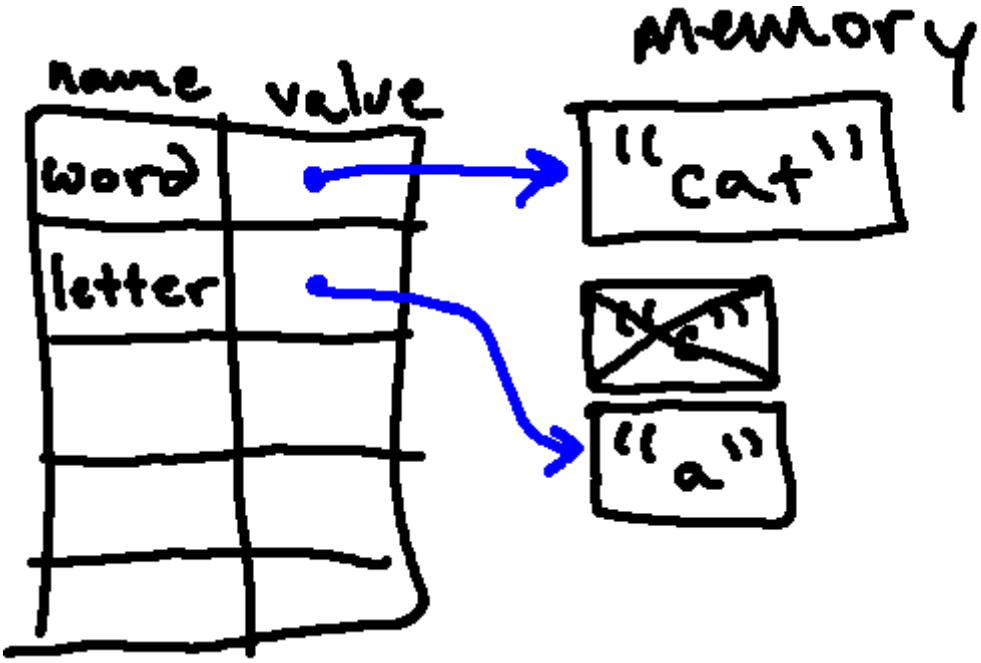
Em seguida, entramos no loop `for`. Antes que a primeira linha no corpo seja executada, Python associa o primeiro caractere de `word` com o nome `letter` (da mesma forma como se tivéssemos feito `letter = word[0]`):



Em seguida, executamos o corpo do loop pela primeira vez. Ao fazer isso, executamos `print(letter)`. Exatamente como faria normalmente, isso envolve a pesquisa do valor de `letter` (que, neste caso, é `"c"`) e a exibição desse valor na tela. Então, quando esta linha é executada, vemos um `c` aparecer na

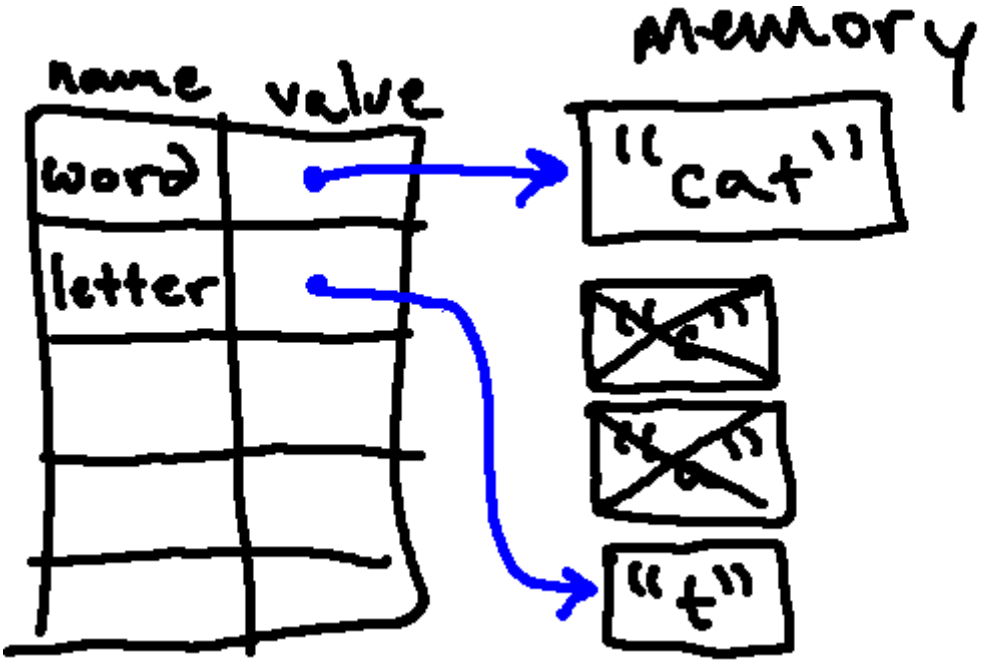
tela.

Agora, como não chegamos ao fim de `word`, vamos fazer um loop novamente. Antes de executarmos o corpo novamente, o Python associa o *próximo* caractere da palavra com o nome `letter` (neste caso, da mesma forma como se tivéssemos feito `letter = word[1]`):



Em seguida, executamos o corpo do loop uma segunda vez. Ao fazer isso, executamos `print(letter)`. Assim como antes, procuramos o valor de `letter`, mas desta vez obtemos `"a"` (já que `letter` foi reatribuído). Então, quando esta linha é executada desta vez, vemos um `a` aparecer na tela.

Agora, como ainda não chegamos ao fim de `word`, vamos fazer um loop novamente. Mais uma vez, antes de executarmos o corpo, o Python associa o próximo caractere de `word` com o nome `letter` (neste caso, da mesma forma como se tivéssemos feito `letter = word[2]`):



Em seguida, executamos o corpo do loop mais uma vez. Ao fazer isso, executamos novamente `print(letter)`. Assim como antes, procuramos o valor de `letter`, mas desta vez obtemos `"t"` (uma vez que `letter` foi novamente reatribuído). Então, quando esta linha é executada desta vez, vemos um `t` aparecer na tela.

Quando terminamos o loop, nosso diagrama de ambiente ainda se parece com este:

e voltamos a executar o resto do programa. Nesse caso, isso significa apenas exibir `'feito'`. Mas se tivéssemos mais código, poderíamos fazer referência a `letter` (que ainda está ligado a `"t"` após sair do loop).

Observe que, como acontece com as instruções `if`, o corpo de um loop for pode conter muitas expressões (todo o código indentado será executado cada vez que passarmos pelo loop).

Observe também que o nome `letter` foi uma escolha arbitrária; podemos usar qualquer nome de variável válido. Portanto, poderíamos ter feito um loop que se comporta exatamente como o anterior, da seguinte maneira:

```
palavra = "cat"
for letra in palavra:
    print(letra)
print('feito')
```

Tente agora:

Tente prever o que o seguinte programa exibirá na tela:

```
# Nota: len(s), onde s é uma string, produz um int cujo valor
# é o número de caracteres na string s fornecida

contagem = 0
palavra = "vaca"
print('Estou pensando em uma palavra.')
print('Tem tantas letras:')
print(len(palavra))
print('Agora vou soletrar a palavra para você.')
for i in palavra:
    print(contagem)
    print(i)
    contagem = contagem + 1
print('A palavra era:')
print(palavra)
print('Que palavra.')
print(contagem)
print(i)
```

Depois de ter uma previsão, digite este código em Python e execute-o.

▶ Mostrar/Esconder

---

3) Outras Sequências

Neste conjunto de leituras, apresentaremos mais dois tipos de sequências: *tuplas* e *listas*.

3.1) Tuplas

Tuplas são sequências como strings, com a importante distinção de que, enquanto as strings são limitadas a conter apenas *caracteres*, tuplas podem conter *objetos arbitrários* (inteiros, floats, booleanos, `None`, ou mesmo outras tuplas!).

As tuplas são especificadas como uma sequência de objetos arbitrários separados por vírgulas, geralmente entre parênteses. Por exemplo, o seguinte é uma tupla contendo três objetos diferentes:

```
x = (7, -7.8, "blue")
```

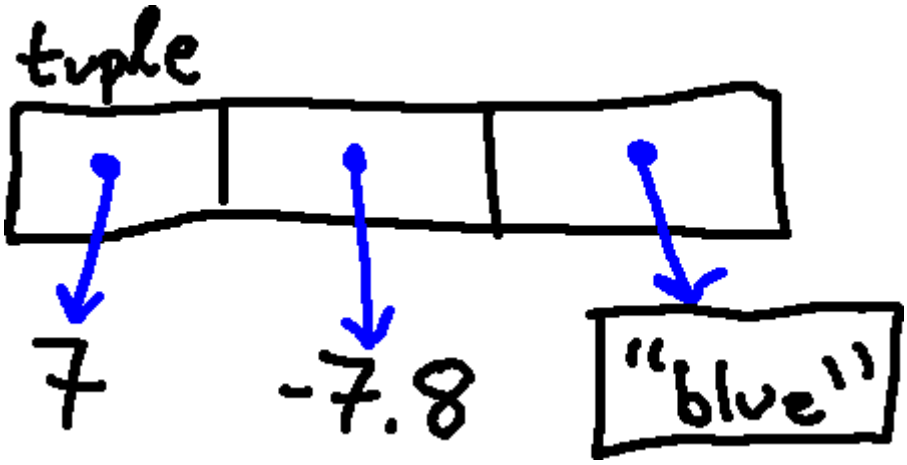
Podemos realizar muitas das mesmas operações em tuplas que podíamos realizar em strings. Por exemplo:

- podemos indexar em uma tupla (`x[1]` nos dá `-7.8`)
- podemos usar `+` para concatenar duas tuplas (`x + (1, 2, 3)` nos dá `(7, -7.8, "blue", 1, 2, 3)`)
- podemos comparar duas tuplas usando `==`
- podemos fazer um loop sobre os elementos em uma tupla usando um loop `for`

Tente agora:

Experimente algumas dessas operações na tupla do exemplo acima, ou com algumas tuplas de sua própria construção.

Também precisamos representar as tuplas em nossos diagramas de ambiente, para modelar como Python realmente as manipula na memória. Vamos modelar a tupla acima `(7, -7.8, "blue")` com o seguinte tipo de desenho:



Vamos desenhá-la como uma caixa, com o rótulo "tupla" (para que possamos rastrear os tipos), com várias referências a outros objetos. Você pode pensar nessas referências como sendo muito semelhantes aos mapeamentos que já consideramos, de nomes a objetos.

Portanto, após avaliar a linha de código acima (`x = (7, -7.8, "blue")`), teremos o seguinte diagrama de ambiente:

~ (https://hz.mit.edu/catsoop/_static/6.145/assignment0.1/readings/tuple2.png)

Vamos examinar o que acontece quando indexamos em `x`. Considere, por exemplo, executar o seguinte código:

```
print(x[-1])
```

Python primeiro procura `x` no quadro global. Fazendo isso, ele segue o ponteiro de `x` e encontra o objeto de tupla na memória. Em seguida, ele procura o índice `-1` dentro de `x`. Este é o último "slot" em `x`, e assim, seguindo esse ponteiro, encontramos a string `"blue"`.

Note que `x[-1]` ainda é uma string e, portanto, qualquer coisa que possamos fazer com qualquer outra string, podemos fazer com `x[-1]`. Isso inclui indexação nele! Então, podemos tentar o seguinte:

```
print(x[-1][2])
```

Ao avaliar `x[-1][2]`, Python irá primeiro procurar `x` (encontrando a tupla na memória). Em seguida, ele irá procurar o índice `-1` dentro dessa tupla (encontrando a string `"blue"`). Finalmente, ele irá procurar o índice `2` dessa string (encontrando `"u"`). Então esta linha acima com imprime um `u` na tela.

Tente agora:

Tente desenhar um diagrama de ambiente para o seguinte código:

```
a = 1
b = 2
c = 3

x = (c, b, a)
y = (3, 2, 1)
```

O que há de diferente em como as duas tuplas são representadas na memória?

▶ Mostrar/Esconder

Tente agora:

Note que as tuplas podem conter *qualquer tipo de objeto Python*, incluindo outras tuplas. Então, poderíamos ter nossa última linha dito: `y = (3, 2, x)`

Como o diagrama de ambiente final seria diferente se fizéssemos essa mudança?

▶ Mostrar/Esconder

Se tivéssemos executado esse código, como Python avaliaria `y[2][0]`?

▶ Mostrar/Esconder

3.2) Listas e mutabilidade

O último tipo de sequência que apresentaremos hoje é um dos tipos integrados mais úteis, a *lista*. Listas são quase iguais a tuplas, com uma exceção que tem grandes consequências potenciais.

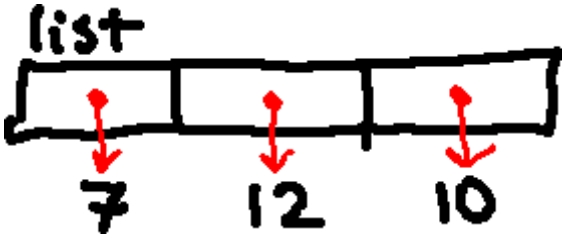
Como strings ou tuplas, listas são sequências. Como tuplas, listas podem conter objetos Python arbitrários. Ao contrário de strings ou tuplas, no entanto, listas são *mutáveis*; isso significa que elas podem ser alteradas após serem criadas. Nesta seção, examinaremos os efeitos dessa diferença.

A sintaxe para criar listas é semelhante à sintaxe usada para criar tuplas, exceto que usa colchetes em vez de parênteses.

Por exemplo, podemos criar algumas listas da seguinte maneira:

```
queijos = ['Cheddar', 'Edam', 'Gouda']
numeros = [42, 123]
outra_lista = [7, 12, 10]
vazia = [] # também podemos fazer uma lista que não contenha elementos!
```

Representaremos listas em diagramas de ambiente de maneira semelhante a como representamos tuplas, mas iremos marcá-las claramente como listas. Por exemplo:



A diferença fundamental é que podemos modificar listas, o que não podíamos fazer com tuplas ou strings. Com uma tupla, poderíamos fazer, por exemplo:

```
minha_tupla = (1,2,3)
print(minha_tupla[0])
```

então procurar itens funciona. Mas se tentássemos modificar um elemento, obteríamos um erro. Então, se tentássemos:

```
minha_tupla[0] = 12
```

veríamos uma mensagem de erro: `TypeError: o objeto 'tupla' não suporta atribuição de item` .

No entanto, se tivéssemos uma lista, poderíamos realmente modificar os elementos contidos na lista!

Tente agora:

Tente executar o seguinte código:

```
queijos = ['Brie', 'Cheddar', 'Mozzarella', 'Americano']
queijos[2] = 'Muenster'
for queijo in queijos:
    print(queijo)
```

O que Python exibe quando executa este código?

▶ Mostrar/Esconder

Tente agora:

Desenhe um diagrama de ambiente para o código a seguir e preveja o que será exibido na tela quando o programa a seguir for executado. Execute seu código para verificar; os resultados podem ser surpreendentes!

```
a = [7, 12, 10]
b = [4, 5, 6]

c = a

print(a)
a[0] = 8
print(a)
print(b)
b[-1] = "cow"
print(b)
print(a)
c[1] = 3.14

for i in a:
    print(i)
```

▶ Mostrar/Esconder

3.3) Adicionando Itens a uma Lista

Outra forma comum de transformar uma lista, além de alterando um dos elementos de uma lista, é *adicionando um nov `element` ao final da lista*. Isso é feito por meio de `append`. Por exemplo, imagine que temos uma lista `x` na memória. Poderíamos então adicionar um `7` ao final dessa lista com a seguinte sintaxe:

```
x = [5, 8, 3, 2, 1]
print(x)
x.append(7)
print(x)
```

Isso irá exibir:

```
[5, 8, 3, 2, 1]
[5, 8, 3, 2, 1, 7]
```

Observe que, ao contrário de concatenar duas listas, isso *não criará uma nova lista*. Em vez disso, ele modificará a lista na memória com a qual `x` já está associado. Embora às vezes tenhamos que ser cuidadosos com isso (devido aos tipos de problemas que vimos acima), modificar uma lista existente na memória é quase sempre substancialmente mais rápido do que fazer uma nova lista por meio de concatenação.

Como listas podem conter objetos Python arbitrários, podemos usar `append` para adicionar qualquer objeto colocando esse objeto entre parênteses em vez de `7`.

```
x.append("uma string!")
x.append((7, 8, 9))
```

Tente agora:

O que será impresso após a execução do seguinte trecho de código?

```
a = [1]
b = a
a.append(6)
a.append(2.0)
a.append("cat")
a[1] = "wolf"
a.append([2])
a = [4]
print(a)
print(b)
```

▶ Mostrar/Esconder

É importante ter em mente, como observamos acima, que `append` funciona modificando a lista associada, em vez de produzir uma nova lista. Também é importante notar que, embora modifique a lista associada, `append` sempre será avaliado como `None`.

Tente agora:

O que acontecerá quando o código a seguir for executado? Preveja o que acontecerá e, em seguida, use Python para verificar sua previsão.

```
a = [7, 4, 8]
b = a.append(9)
print(a)
print(b)
c = b.append(9)
print(a)
print(b)
print(c)
```

▶ Mostrar/Esconder

3.3.1) Padrão Comum: Construindo Listas Relacionadas

Um padrão comum envolve o uso de `append` para construir uma lista de valores com base em alguma outra lista. Por exemplo, imagine que temos uma lista de inteiros e queremos criar uma lista dos quadrados dos números pares na lista original. Poderíamos fazer isso com, por exemplo, o seguinte código:

```
original_list = [7, 4, 8, 2, 9]
# primeiro faça uma lista vazia para conter os resultados
new_list = []
# para cada número na lista original, faça o seguinte:
for element in original_list:
    if element % 2 == 0: # se o número for par ...
        # adiciona seu quadrado à nova lista
        new_list.append(element ** 2)
print(new_list)
```

Tente agora:

Use um diagrama de ambiente para ajudar a percorrer o código e para explicar que valor será impresso na tela na última linha.

▶ Mostrar/Esconder

4) Padrão Comum: Loop sobre Certos Valores Inteiros

Nas seções acima, introduzimos a noção de loop em uma sequência usando a palavra-chave `for`. Em um loop `for`, realizamos um cálculo uma vez para cada elemento em uma sequência, definindo uma variável específica para apontar para cada valor por vez.

Às vezes, você deseja fazer um loop sobre valores específicos para os quais construir uma sequência seria uma dor. Por exemplo, considere imprimir os quadrados de todos os inteiros de 0 a 24. Uma maneira de fazer isso seria escrever o seguinte:

```
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19, 20, 21, 22, 23, 24]:
    print(i ** 2)
```

mas escrever a lista de valores que estamos repetindo neste caso é uma verdadeira dor! Convenientemente, Python nos oferece uma maneira mais fácil de construir estruturas dessa forma com `range`. `range` pode ser usado para produzir um tipo especial de objeto `range`, que, embora não seja exatamente o mesmo que uma lista ou tupla, pode servir ao mesmo propósito quando usado como parte de um loop `for`.

Para fazer o acima de forma mais compacta, poderíamos ter escrito:

```
for i in range(25):
    print(i ** 2)
```

Se você invocar `range` com `range(4)`, ele fornecerá `4` elementos, começando de 0 e contando para cima. Se você invocá-lo com `range(7)`, ele fornecerá sete elementos. Quando usado desta forma, `range` terá o mesmo propósito que uma lista contendo `[0, 1, 2, 3]` ou `[0, 1, 2, 3, 4, 5, 6]`.

O loop pela sequência de inteiros é uma ocorrência bastante comum e, portanto, ter `range` disponível pode realmente nos poupar muito de digitação (particularmente se a sequência de inteiros que você deseja fazer o loop for longa!).

4.1) Padrão Comum: Loop sobre os Índices em uma Lista'

Outro padrão comum envolve o loop nos valores de uma lista. Se quiséssemos, porém, poderíamos usar nossa variável de loop para conter os *índices* de uma lista. Tente executar o alcance dos seguintes programas. O que cada um imprime e por quê?

```
entradas = ['cachorro', 'gato', 'furão', 'hamster']
for i in entradas:
    print(i)
```

```
entradas = ['cachorro', 'gato', 'furão', 'hamster']
for i in range(len(entradas)):
    print(i)
```

Tente agora:

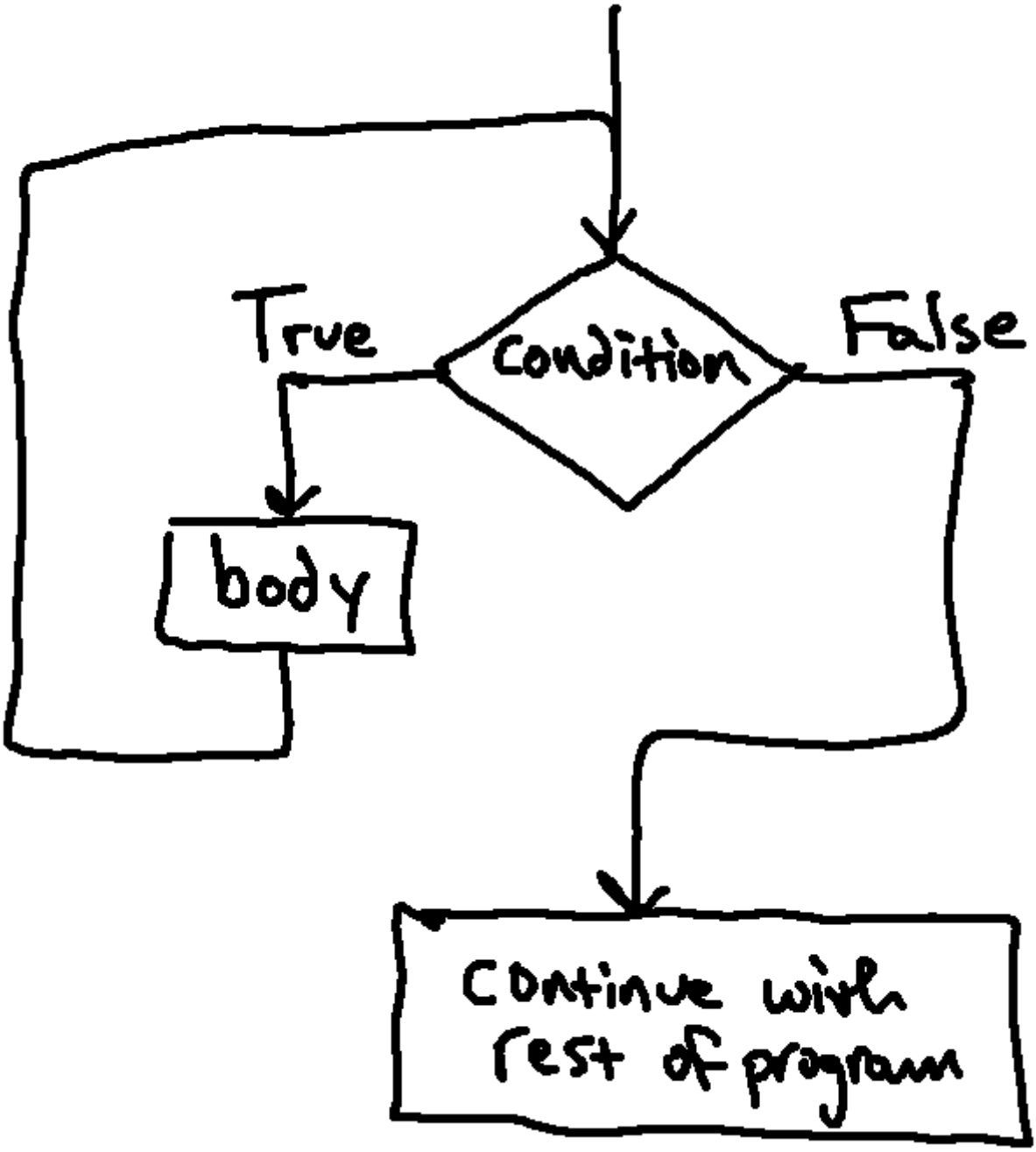
Se quiséssemos o índice *e* o valor disponíveis dentro do corpo do nosso loop, como poderíamos fazer isso?

▶ Mostrar/Esconder

5) Looping até que uma Condição seja Satisfeita

Muito da nossa discussão neste conjunto de leituras foi sobre *iteração*, que é a capacidade de executar um bloco de instruções repetidamente. Anteriormente, vimos que, usando `for`, tínhamos uma maneira de executar um bloco de instruções uma vez para cada elemento em uma sequência. Isso era muito conveniente, mas às vezes não conhecemos uma sequência particular de elementos sobre a qual desejamos iterar, nem quantas vezes gostaríamos de executar o loop. Às vezes, queremos repetir uma sequência de declarações até que uma condição particular seja satisfeita.

Para esse fim, o Python oferece outra construção de loop, um loop `while`. Um loop `while` é muito parecido com um condicional, no sentido de que consiste em uma condição e em um corpo e usa a condição para decidir se deve executar o corpo ou pulá-lo. A diferença é: enquanto uma condicional executa o corpo exatamente uma vez e segue em frente, um loop `while` continuará executando o corpo até que a condição não seja mais avaliada como `True`. Este padrão de fluxo é representado neste fluxograma:



Primeiro entramos neste diagrama a partir do topo. Se a condição for avaliada como `False`, então pulamos o loop totalmente e seguimos em frente, mas se for `True`, entramos no corpo do loop. A diferença de uma condicional regular é que, se executarmos o corpo, então, quando terminarmos, voltaremos e verificamos a condição novamente (em vez de continuar). Se a condição for novamente `True`, entraremos no loop novamente e assim por diante.

Considere o seguinte exemplo:

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

Você quase pode ler a instrução `while` como se fosse em palavras. Significa: "Enquanto `n` for maior que 0, exiba o valor de `n` e então diminua `n` de 1. Quando chegar a 0, exiba a palavra `Blastoff!`"

Um pouco mais formalmente, aqui está o fluxo de execução para uma instrução `while`:

1. Determina se a condição é verdadeira ou falsa.
2. Se for falso, saia da instrução `while` e continue a execução na próxima instrução.
3. Se a condição for verdadeira, execute o corpo e volte para a etapa 1. Assim como escrito, o programa acima imprimirá:

```
5
4
3
2
1
Blastoff!
```

Tente agora: Por que `0` não foi impresso quando o programa foi executado? Como você poderia modificá-lo para que imprimisse um `0` também, antes de imprimir o "blastoff"?

▶ Mostrar/Esconder

Tente agora:

O que teria sido impresso se tivéssemos definido `n = -1` em vez de `n = 5` na primeira linha?

▶ Mostrar/Esconder

É importante, ao escrever loops `while`, certificar-se de que o corpo do loop altere o valor de uma ou mais variáveis para que a condição se torne falsa eventualmente e o loop termine. Caso contrário, o loop se repetirá para sempre, o que é chamado de *loop infinito*.

No caso do programa de contagem regressiva, podemos provar que o loop termina: se `n` for zero ou negativo, o loop nunca será executado. Caso contrário, `n` fica menor a cada vez que passa pelo loop, então, eventualmente, temos que chegar a 0.

Para alguns outros loops, não é tão fácil dizer. Por exemplo:

```
n = 27
while n != 1:
    print(n)
    if n % 2 == 0: # n é par
        n = n / 2
    else: # n é ímpar
        n = n * 3 + 1
```

A condição para este loop é `n != 1`, então o loop continuará até que `n` seja `1`, o que torna a condição falsa.

Cada vez que passa pelo loop, o programa produz o valor de `n` e, a seguir, verifica se ele é par ou ímpar. Se for par, `n` é dividido por 2. Se for ímpar, o valor de n é substituído por `n * 3 + 1`. Por exemplo, se `n` começa como `3`, os valores resultantes de n são 3, 10, 5 , 16, 8, 4, 2 e 1.

Visto que `n` às vezes aumenta e às vezes diminui, não há nenhuma prova óbvia de que `n` chegará a 1 ou de que o programa termina. Para alguns valores particulares de `n`, podemos provar a terminação. Por exemplo, se o valor inicial for uma potência de dois, `n` será pae a cada vez através do loop até atingir 1. O exemplo anterior termina com tal sequência, começando com 16.

A pergunta difícil é se podemos provar que esse programa termina para *todos* os valores positivos de `n`. Até agora, ninguém foi capaz de provar ou contestar! (veja página da Wikipedia para a conjectura de Collatz (https://pt.wikipedia.org/wiki/Conjectura_de_Collatz))

Tente agora:

Qual sequência de valores seria impressa pelo loop acima se tivéssemos começado com `n = 6`? Simule manualmente primeiro e depois use Python para testar!

▶ Mostrar/Esconder

5.1) Exemplo: Aproximando Raízes Quadradas

Loops são frequentemente usados em programas que calculam resultados numéricos começando com uma resposta aproximada e melhorando iterativamente.

Por exemplo, uma maneira de calcular raízes quadradas é o método de Newton. Suponha que você queira saber a raiz quadrada de a. Se você começar com quase qualquer estimativa, x, poderá calcular uma estimativa melhor com a seguinte fórmula:

$$y = \frac{x + a/x}{2}$$

Por exemplo, se a é 4 e x é 3:

```
a = 4
x = 3
y = (x + a / x) / 2
print (y) # exibe 2.16666666667
```

O resultado está mais próximo da resposta correta ($\sqrt{4} = 2$). Se repetirmos o processo com a nova estimativa, chegamos ainda mais perto:

```
x = y
y = (x + a / x) / 2
print (y) # exibe 2.00641025641
```

Depois de mais algumas atualizações, a estimativa é quase exata:

```
x = y
y = (x + a / x) / 2
print (y) # exibe 2.00001024003

x = y
y = (x + a / x) / 2
print (y) # exibe 2.00000000003
```

Em geral, não sabemos com antecedência quantas etapas são necessárias para chegar à resposta certa, mas sabemos quando chegamos lá porque a estimativa para de mudar:

```
x = y
y = (x + a / x) / 2
print (y) # exibe 2.0
x = y
y = (x + a / x) / 2
print (y) # exibe 2.0
```

Quando `y == x`, podemos parar. Aqui está um loop que começa com uma estimativa inicial, `x`, e a melhora até que pare de mudar:

```
a = 4
x = None
y = 2.5
while x != y:
    x = y
    print(x)
    y = (x + a / x) / 2
```

Para a maioria dos valores de `a`, isso funciona bem, mas em geral é perigoso testar a igualdade de `float` (por algumas das razões que falamos na última seção, especificamente que os floats não podem representar com precisão todos os números!).

Em vez de verificar se `x` e `y` são exatamente iguais, é mais seguro fazer um loop até que a diferença entre eles ou a diferença entre eles se torne pequena o suficiente (comparando com uma pequena margem de erro).

6) Loops aninhados

O corpo de um loop (a parte indentada) pode conter código Python arbitrário. Isso significa que, entre outras coisas, é possível que o próprio corpo de um loop contenha uma condicional ou um loop!

Tente agora:

Considere o seguinte exemplo:

```
for i in range(9):
    for j in range(9):
        print((i + 1) * (j + 1))
```

Observe que o loop `for` interno (aquele com `j`) está completamente contido no loop `for` externo (aquele com `i`). Preveja a saída deste programa e, em seguida, use Python para verificar sua previsão.

▶ Mostrar/Esconder

Tente agora:

Considere o seguinte trecho de código, que é projetado para fazer um loop em uma lista Python, coletando elementos positivos até encontrar `0` (ignorando os números negativos antes desse ponto e ignorando *todos* os valores depois de `0`). No entanto, conforme está escrito, ele não atinge seu objetivo! O que acontece quando você digita o seguinte em Python e o executa? Qual é o erro e como pode ser corrigido?

```
entradas = [7, 8, -2, -1, 8, 9, 0, 12, 4, -8, 2]
saidas = [] # deve ser [7, 8, 8, 9] quando terminarmos
print('começando agora')
for element in entradas:
    while elemento != 0:
        if elemento > 0:
            saidas.append(elemento)
print('pronto!')
print(saidas)
```

▶ Mostrar/Esconder

Tente agora:

Agora que entendemos o problema com o código acima, como podemos corrigi-lo? Você consegue pensar em uma maneira de atingir a meta declarada? Tente modificar ou reescrever o código acima para corrigir o problema.

▶ Mostrar/Esconder

---

7) Debugging

Neste conjunto de leituras, introduzimos algumas estruturas novas e começaremos a nos mover em direção a programas mais complicados, que podem ser mais difíceis de imaginar. Em geral, podemos tentar gerenciar essa complexidade tentando primeiro quebrar nossos programas em pequenos pedaços, que podem ser escritos e testados independentemente dos outros (isso é conhecido como *design modular* porque estamos pensando em dividir o programa em *módulos* separáveis). Geralmente é muito mais fácil planejar, testar e implementar partes individuais conforme você avança, em vez de passar horas escrevendo um grande programa e depois descobrir que ele não funciona e ter que vasculhar todo o seu código, tentando encontrar os bugs.

No entanto, mesmo com todo o design inteligente do mundo, você ainda ocasionalmente se encontrará na posição (inevitável) de ter um grande programa com um bug nele; nesse caso, não se desespere! Debugging de um programa não requer brilhantismo, criatividade ou muito conhecimento. O que requer é persistência e uma abordagem sistemática, porque requer raciocínio não apenas sobre o que queremos, mas sobre como o Python se comportará em resposta aos nossos programas (por isso é tão importante ter um modelo mental forte de Python!).

Em primeiro lugar, é crucial ter um caso de teste em mente (um conjunto de entradas para o programa que você está tentando depurar) e *saber qual deve ser a resposta*, tanto para o programa geral quanto para valores intermediários relevantes. Para encontrar um bom caso de teste, você pode começar com alguns casos especiais: e se o argumento for 0 ou a lista vazia? E se for negativo? Esses casos podem ser mais fáceis de resolver primeiro (e também são casos que podem estar facilmente errados). Em seguida, tente casos mais gerais.

Para a maioria dos programas deste curso, você deveria simular seu código manualmente usando um diagrama de ambiente antes de executá-lo em Python. Isso é tedioso, sim, mas realmente é importante para ajudá-lo a construir um modelo mental forte de como o Python se comporta. Com mais experiência, você será capaz de fazer essas previsões rapidamente em sua cabeça. Mas, por enquanto, *desenhe!*

Então, a questão permanece: se o seu programa der errado em seus casos de teste, o que você deve fazer? **Resista à tentação de começar a mudar seu programa, apenas para ver se isso resolverá o problema.** Não altere nenhum código até saber o que está errado com o que está fazendo agora e, portanto, acredite que a alteração que fizer corrigirá o problema.

Já temos algumas ferramentas à nossa disposição para esse fim, que podem funcionar razoavelmente bem para pequenos programas: o modelo de substituição para avaliação de expressões e diagramas de ambiente. O ato de simular com essas ferramentas pode ajudá-lo a encontrar o seu erro (embora seja importante lembrar que o Python não sabe o que você quer fazer, apenas o que você diz a ele para fazer; por isso, é importante ser sistemático ao seguir seu código).

Às vezes, você pode não conseguir encontrar seu bug no papel. Para esses casos, o método que mostraremos se concentra em depurar sistematicamente usando instruções `print` . Hoje em dia existem outras ferramentas além de `print` para ajudar em debugging (logicamente chamadas de *debuggers*), mas é raro para muitos (mesmo após anos de experiência em programação) que essas ferramentas sejam necessárias. Em minha opinião, `print` ainda é a ferramenta de depuração mais direta, mais poderosa e mais geral que existe.

Uma boa maneira de usar as instruções `print` para ajudar na depuração é usá-las para exibir os resultados das etapas intermediárias ao longo do caminho. Dependendo da estrutura do seu programa, isso pode ser: os valores que você está percorrendo (para ter certeza de que seus limites estão corretos), uma solução completa para um subproblema, uma solução parcial para o problema geral. Para os locais

escolhidos, você deve exibir *tanto a quantidade de interesse quanto o valor que espera que essa quantidade tenha*. Se forem iguais, pode ser que essa parte do código esteja funcionando corretamente e você pode tentar `print` ing em outros locais.

Uma estratégia aqui é usar uma variação da pesquisa binária. Encontre um ponto mais ou menos na metade do seu código no qual você possa prever os valores das variáveis ou resultados intermediários do seu cálculo. Coloque uma declaração `print` lá que liste os valores esperados e reais das variáveis. Execute seu caso de teste e verifique. Se os valores previstos corresponderem aos reais, é provável que o bug ocorra após este ponto no código; se não, então você tem um bug anterior a este ponto (é claro, você pode ter um segundo bug após este ponto, mas você pode descobrir isso mais tarde). Agora repita o processo encontrando um local a meio caminho entre o início do procedimento e este ponto, colocando uma declaração de impressão com os valores esperados e reais e continuando. Desta forma, você pode restringir a localização do bug. Estude essa parte do código e veja se você pode ver o que está errado. Caso contrário, adicione mais algumas instruções `print` perto da parte problemática e execute-o novamente.

A regra mais importante de debugging é: **Não tente ser inteligente .... seja sistemático e infatigável!**

8) Resumo

Nesta leitura, começamos a expandir as ideias introduzidas no início da semana, introduzindo três novos tipos de *objetos compostos* (strings, tuplas e listas) e vendo como eles se encaixam no modelo mental que começamos a construir da última vez, e também introduzindo novas maneiras de controlar a ordem em que Python executa as instruções (`for e while`).

Neste conjunto de exercícios, você obterá alguma prática com essas novas peças, bem como uma revisão das peças mais antigas. No próximo conjunto de leituras e exercícios, apresentaremos um meio realmente poderoso de abstração: funções