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) Como Usar Essas Notas

Ao aprender a programar, não basta apenas ouvir ou ler; você precisa praticar! Neste curso, as leituras apresentarão novas ideias, mas também solicitarão que você participe!

Haverá momentos nesta página em que você será solicitado a tentar coisas por conta própria (algumas delas podem ser "checadas" online pelo nosso sistema, mas algumas são apenas atividades sugeridas. De qualquer forma, é importante fazer (e *entender*) todas essas atividades! Se você está tendo problemas para entender por que algo se comportou daquela forma (mesmo se você foi capaz de produzir o resultado esperado), peça ajuda!

Nas notas, ocasionalmente também será solicitado que você digite um programa em Python. Nesses casos, realmente recomendamos que você digite o programa em Python literalmente, em vez de usar, por exemplo, copiar e colar. Pode parecer uma coisa pequena (e talvez um pouco tediosa), mas é importante mesmo assim! Você descobrirá rapidamente que Python é muito meticuloso quanto às entradas que aceita; o ato de digitar o programa caractere por caractere, observando os erros se eles existirem e comparando o código com o que é mostrado nesta página é uma boa preparação para o trabalho orientado a detalhes que é programação!

Estamos sempre aqui para ajudar, mas como Morpheus disse:

> *"Eu só posso mostrar a porta. Você é quem tem que passar por ela."*
> *-Laurence Fishburne como Morpheus, The Matrix*

Tentaremos fornecer todos os materiais e suporte de que você precisa, mas cabe a você fazer uso deles (e também nos informar sobre a melhor forma de ajudá-lo!).

1) Instalando Python

Você precisará de Python 3.7 ou posterior para acompanhar os materiais e problemas do curso. Se você já tiver Python instalado e acessível, ótimo! Se não, esta seção contém instruções para configurar seu computador com os recursos necessários para o curso.

Windows (64-bit)

▶ Mostrar/Esconder

Mac OS

▶ Mostrar/Esconder

2) IDLE, e o seu Primeiro Programa

Independente do seu sistema operacional, Python deve ter vindo com um ambiente de desenvolvimento integrado chamado IDLE. Se você já tiver algum editor de texto de preferência, fique à vontade para usá-lo. A recomendação é usar IDLE por ser (relativemente) simples e ser feito para uso com Python.

Abra o IDLE agora (deve aparecer como qualquer outro aplicativo) para escrevermos nosso primeiro programa! Abra um novo arquivo (`Ctrl+N` ou `⌘-N`). Uma nova janela vai aparecer com um editor de texto em branco. Chegou a hora de escrever nosso primeiro programa e ver todo o poder de programar em Python. Escreva o seguinte (*exatamente*) na janela nova:

```
print(2 + 3)
```

Talvez não seja tão grandioso assim, mas um começo é um começo. Esse programa contém uma única instrução `print` , que faz Python mostrar um valor na tela (vamos voltar a isso depois).

O resto é razoavelmente direto. Quando executarmos, o programa somará os números `2` e `3` e mostrará o resultado.

Para executar o programa, você precisará primeiro salvá-lo (`Ctrl+S` ou `⌘-S`) e dar-lhe um nome (qualquer nome funcionará por enquanto, mas arquivos Python devem terminar em `.py`). Uma vez salvo, aperte `F5` (ou ache "Run Module" no menu "Run"). Isso levará de volta à janela principal do IDLE e você conseguirá ver of resultado de executar seu programa.

```
print(2 + 3)
```

Tente agora:

Agora tente remover os parênteses da instrução `print` para que fique da forma

```
print 2 + 3
```

Tente salvar e executar o programa. O que acontece?

▶ Mostrar/Esconder

---

3) The Way of The Program

O objetivo final deste curso, de certa forma, é ensinar você a pensar de uma maneira particular, como um programador experiente faz. Essa forma de pensar combina algumas das melhores características da matemática, engenharia e ciências naturais. Como os matemáticos, os programadores usam linguagens formais para descrever ideias (especificamente computações). Como engenheiros, eles projetam coisas, montando componentes em sistemas e avaliando as melhores alternativas. Como cientistas, eles observam o comportamento de sistemas complexos, formam hipóteses e testam previsões.

A habilidade mais importante para um programador é a resolução de problemas. Resolver problemas significa a capacidade de formular problemas, pensar criativamente sobre as soluções e expressar uma solução de forma clara e precisa. O processo de aprender a programar é uma excelente oportunidade para praticar habilidades de resolução de problemas!

Por um lado, você aprenderá a programar em Python, uma habilidade útil por si só. Por outro, você usará a programação como um meio para um fim, para resolver um problema. À medida que avançamos, esse fim se tornará mais claro.

3.1) O que é um programa?

Um *programa* (às vezes chamado de *script*) é uma sequência de instruções que especifica como realizar uma computação. O cálculo pode ser algo matemático, como resolver um sistema de equações ou encontrar as raízes de um polinômio, mas também pode ser uma computação simbólica, como pesquisar e substituir texto em um documento, ou algo gráfico, como processar uma imagem ou editar um vídeo.

Os detalhes parecem diferentes em diferentes linguagens de programação, mas algumas instruções básicas aparecem em quase todas:

- **input**: informação obtida do teclado, de um arquivo, de uma rede, ou algum outro dispositivo.
- **output**: mostrar dados na tela, salvá-los em um arquivo, enviá-los por uma rede, etc.
- **matemática**: realizar operações matemáticas básicas como adição, multiplicação.
- **execução condicional**: checar certas condições e executar código apropriadamente.
- **repetição**: realizar alguma ação repetidamente, geralmente com alguma variação.

Isso é basicamente tudo que há. Qualquer programa que você já tenha usado, não importa quão complicado, é feito de instruções que parecem muito com essas. Programação nada mais é do que o processo de quebrar uma tarefa grande, complexa, em subtarefas menores e menores até que estas sejam pequenas o suficiente para ser concluída por uma dessas instruções básicas.

4) Nossos Objetivos

Para trabalhar de forma eficaz, você precisará ter um entendimento razoável de como Python responderá a diferentes entradas. Passaremos a maior parte do tempo ao longo do curso tentando desenvolver um modelo mental apropriado do Python, para que possamos ler, escrever e *debug* nossos programas de forma mais efetiva.

É importante notar que nem sempre trabalharemos com modelos perfeitos: às vezes faremos simplificações no começo e voltaremos depois para preenchermos "buracos" ou para fazer correções.

Durante o curso, estaremos desenvolvendo um kit de ferramentas e uma maneira de pensar que o capacitará a criar seus próprios programas (a partir desses mesmos blocos básicos de construção) e a compreender programas em Python (tanto seus quanto aqueles escritos por outros).

Vamos começar agora com algumas definições.

5) Objetos, Valores e Tipos Primitivos

Objetos são as principais coisas com que programas em Python trabalham (na falta de uma melhor descrição). Você já trabalhou com alguns objetos de Python (especificamente `2` e `3`) no programa acima.

Cada objeto tem um *tipo* e um *valor*: o valor do objeto determina a coisa exata que ele representa, e seu tipo determina que tipo de coisas programas podem fazer com esse objeto.

Frequentemente nos referiremos aos tipos como *primitivos* ou *compostos*. Os tipos primitivos são atômicos: são indivisíveis e, em certo sentido, os menores pedaços da linguagem. Objetos compostos, como você pode adivinhar, são compostos de um ou mais tipos primitivos.

Dependendo de como você contar, Python tem cerca de quatro tipos primitivos, a maioria dos quais representa diferentes tipos de números:

1. `int` é um tipo usado para representar números inteiros. Você já viu dois inteiros no seu primeiro programa: `2` e `3`. Inteiros em Python são escritos do jeito que normalmente escrevemos inteiros (o único detalhe é que não podemos usar vírgulas!). Por exemplo, `2`, `3`, `0`, `-12`, `10000` são todos `int`.
2. `float` é um tipo usado para representar números reais. A maneira mais comum de especificar um `float` é como um número com um ponto decimal. Note que decimais são marcados com ponto final (`2.5` ao invés de `2,5`) pelo padrão estadunidense! Qualquer número com um ponto decimal é um `float` (por exemplo, `5.0` ou `6.2831` ou `3.`). Existem algumas outras maneiras de especificar `float`s, mas vamos guardá-las para mais tarde.

   Você pode se perguntar por que esse tipo não é chamado de "real" ou "exato" ou "decimal" ("float" parece bem bizarro, de fato!). A razão é que esses valores são representados em um computador em uma representação chamada *floating point* ("ponto flutuante"). Não entraremos em detalhes sobre essa representação nesta aula, mas vale a pena observar algumas coisas:

   - Essa representação é usada por quase todas as linguagens de programação modernas, já que ela tem alguns recursos interessantes.
   - No entanto, é importante notar que esta representação não consegue representar exatamente todos os números reais. Às vezes precisaremos ter em mente que os objetos `float` são *aproximações* dos números que realmente queremos.
   - Para ver um exemplo desse comportamento, modifique seu programa original para que, em vez de mostrar `2 + 3`, mostre `0.1 + 0.1 + 0.1`. Salve e execute seu programa e você verá um resultado inesperado!
3. `bool` é um tipo usado para representar valores booleanos. Em Python, estes são representados como `True` e `False` (note que precisam ser capitalizados! `true` não é um booleano).
4. `NoneType` é um tipo com um único valor: `None`. `None` é um tipo especial de objeto que é usado para representar a *ausência* de um valor. Isso pode parecer meio estranho agora, mas deve fazer mais sentido com tempo.

5.1) Convertendo entre Tipos Numéricos

Às vezes é possível converter objetos a algum outro tipo. Por exemplo, tente `int(7.8)` ou `float(6)`. Converter um `float` a um `int` talvez não faça o que esperaríamos (remove toda a parte decimal diretamente). Python também fornece uma maneira de arrendodar números da maneira tradicional, por exemplo usando `round(7.8)`.

6) Expressões Algébricas

Como vimos antes, não estamos limitados a trabalhar com esses tipos básicos apenas: podemos combiná-los usando *operadores*. Por enquanto, você já viu o seu primeiro operador usado com inteiros: `+` .

Tente agora: Modifique seu programa para que leia:

```
print(2 + 3)
print(7 - 3)
```

Antes de executá-lo, o que você espera que seja mostrado? Uma vez que tenha pensado nisso, salve e execute o programa.

▶ Mostrar/Esconder

Você já deve imaginar que Python consegue fazer muito mais que adição; também consegue subtrair! Na verdade, existem mais algumas operações que devemos conhecer agora:

- `+` representa adição.
- `-` representa subtração.
- `*` representa multiplicação.
- `/` representa divisão.
- `i ** j` representa exponenciação (**cuidado!** Outras linguagens usam `^` para exponenciação, mas em Python esse símbolo tem significado completamente diferente!)

Tente agora:

Modifique seu programa para que mostre um resultado para cada tipo de operadores acima.

Tente agora:

Vamos introduzir dois novos operadores, ambos relacionados a divisão: `//` e `%`. Tente escrever algumas expressões simples usando esses operadores no seu programa para descobrir o que eles significam.

▶ Mostrar/Esconder

Tente agora:

Antes, percebemos que objetos em Python têm um tipo e um valor. Vimos acima como operações produzem um valor resultante. Mas existem também regras associadas a que *tipo* de resultado uma operação tem.

Tente experimentar com diferentes tipos: troque alguns dos objetos `int` com `float`s no seu programa. Você consegue descobrir as regras que determinam o tipo do resultado?

▶ Mostrar/Esconder

O que acontece se você tentar combinar um `NoneType` com um `int` ou `float` usando os operadores acima?

▶ Mostrar/Esconder

---

7) Modelo de Interpretação de Expressões

Python não é limitado a expressões com um único operador. Também é possível usar expressões mais complicadas, como:

```
2 * 3 ** 2 + 4
```

Como Python avalia uma expressão desse tipo?

7.1) Ordem de Operações

Quando uma expressão contém mais de um operador, a ordem de avaliação depende da *ordem das operações*. Para operadores matemáticos, Python segue a convenção matemática. A sigla "PEMDAS" é uma maneira útil de lembrar as regras:

- Os **P**arênteses têm a precedência mais alta e podem ser usados para forçar uma expressão a ser avaliada na ordem desejada. Visto que as expressões entre parênteses são avaliadas primeiro, `2 * (3-1)` é `4` e `(1+1) ** (5-2)` é `8`. Você também pode usar parênteses para tornar uma expressão mais fácil de ler, como em `(2 * 100) / 60`, mesmo que não altere o resultado.
- A **E**xponenciação tem a próxima precedência mais alta, então `1 + 2 ** 3` é `9` (não `27`) e `3 ** 2 * 2` é `18` (não `81`).
- **M**ultiplicação e **D**ivisão têm precedência mais alta do que **A**dição e **S**ubtração. Portanto, `2*3-1` é `5` (não `4`) e `6+4/2` é `8.0` (não `5.0`).

Operadores com a mesma precedência são avaliados da esquerda para a direita (exceto exponenciação). Portanto, na expressão `7/2 * 3`, a divisão acontece primeiro e o resultado é multiplicado por `3`. Para dividir por `2*3`, você pode usar parênteses ou escrever `7 / 2 / 3`.

Não é necessário muito trabalho para lembrar a precedência dos operadores. Se não puder dizer olhando para a expressão, sempre podemos usar parênteses para tornar mais clara a ordem em que as coisas estão acontecendo.

7.2) O Modelo de Substituição

Nosso primeiro modelo de avaliação de expressão é o *modelo de substituição*: quando Python avalia uma expressão, ele trabalha na ordem das operações, avaliando as subexpressões e substituindo cada uma por seu resultado. Por exemplo, considere avaliar a expressão `2 * 3 ** 2 + 4.0`

Python respeitará a ordem das operações, primeiro avaliando `3**2` para encontrar `9` e substituindo esse valor na expressão original. Após esta etapa, agora temos: `2 * 9 + 4.0`.

Novamente seguindo a ordem das operações, o Python avaliará em seguida `2*9` para encontrar `18` e substituirá esse valor na expressão. Após esta etapa, temos: `18 + 4.0`.

Finalmente, o Python avalia `18+4.0` para encontrar `22.0`, que é o resultado da avaliação de toda a expressão.

Tente agora:

Use o modelo de substituição para entender passo-a-passo como Python avalia `7 + 8 * 6 / 2 ** 3`.

▶ Mostrar/Esconder

---

8) Expressões Booleanas

Uma *expressão booleana* é uma expressão que é ou verdadeira ou falsa, e é avaliada a um objeto do tipo `bool` cujo valor é `True` ou `False`.

Os exemplos a seguir usam o operador `==`, que compara dois operandos e produz `True` se eles são iguais ou `False` caso contrário:

```
print(5 == 5)
print(5 == 6)
```

Este exemplo mostrará `True` em uma linha e `False` na próxima.

O operador `==` é um dos *operadores relacionais* (que operam em quaisquer valores e produzem objetos `bool`). Estes operadores são:

- `==` ("é igual a") compara dois operandos e produz `True` se eles forem iguais e `False` caso contrário.
- `!=` ("não é igual a") compara dois operandos e produz `True` se eles não forem iguais e `False` caso contrário.
- `>` ("é maior que") compara dois operandos e produz `True` se o primeiro for maior que o segundo e `False` caso contrário.
- `<` ("é menor que") compara dois operandos e produz `True` se o primeiro for menor que o segundo e `False` caso contrário.
- `>=` ("é maior ou igual a") compara dois operandos e produz `True` se o primeiro for maior ou igual ao segundo, e `False` caso contrário.
- `<=` ("é menor ou igual a") compara dois operandos e produz `True` se o primeiro for menor ou igual ao segundo, e `False` caso contrário.

Embora essas operações provavelmente sejam familiares para você, os símbolos em Python são diferentes dos símbolos matemáticos. Um erro comum é usar um único sinal de igual (`=`) em vez de um duplo sinal de igual (`==`). `=` tem um significado diferente em Python, que discutiremos na próxima seção.

Existem também três operadores que operam somente em valores booleanos:

- `and` produz `True` se ambos os operandos forem `True` e produz `False` caso contrário (por exemplo, `True and True` produz `True`, mas `False and True` produz `False`). Inclui *short-circuiting*: se algum operando for `False`, os próximos operandos não são sequer avaliados (já resulta em `False`).
- `or` produz `True` se pelo menos um de seus operandos for `True`, e produz `False` caso contrário (por exemplo, `True or True` produz `True`; e `True or False` também produz `True`; mas `False or False` produz `False`). Inclui *short-circuiting*: se algum operando for `True`, os próximos operandos não são sequer avaliados (já resulta em `True`).
- `not` é um operador *unário* (tem apenas um operando) que produz `True` se seu operando for `False`, e `False` se seu operando for `True` (por exemplo, `not False` produz `True`; e `not True` produz `False`).

8.1) Modelo de Substituição Completo

Expressões em Python podem ser bastante complicadas e conter qualquer número dos operadores descritos acima (incluindo tanto operadores booleanos e aritméticos).

A ordem de avaliação incluindo booleanos é baseada em:

- Operações aritméticas vêm primeiro, na ordem descrita acima.
- Depois os operadores `==`, `!=`, `<`, `>`, `<=`, `>=` são avaliados. Todos têm a mesma precedência, então são avaliados da esquerda para a direita.
- Depois o operador `not` é avaliado.
- Depois `and`.
- Finalmente, `or`.

Tente agora:

Como um exemplo, tente seguir como Python avaliaria a seguinte expressão usando o modelo de substituição:

```
7**2*2 > 97 or 2*3+90 < 10*(5+4) and not 20 == 4 * 6
```

▶ Mostrar/Esconder

---

9) Variáveis e Atribuição

Um dos recursos mais poderosos de uma linguagem de programação é a capacidade de manipular *variáveis*. Uma variável é um nome que se refere a um valor.

Criamos uma nova variável e a associamos a um valor com uma instrução de *atribuição*, que faz uso do *operador de atribuição* (o sinal de "igual", `=`). Por exemplo, o pequeno trecho de código a seguir associa o nome `x` ao valor `4`.

```
x = 2 + 2
```

É importante notar que este operador não funciona como o sinal de "igual" em álgebra. Python responderá a esse tipo de instrução avaliando a expressão no lado direito do operador e, em seguida, associando o nome no lado esquerdo do operador com o valor resultante.

Portanto, esta declaração não está realmente dizendo "x é igual a $2 + 2$", pelo menos no sentido que você deve estar familiarizado com a matemática. Em vez disso, o que realmente diz ao Python é: calcule o valor `2+2` e associe o nome `x` ao resultado.

Uma vez que Python tem essa associação, podemos incluir `x` em uma expressão, assim como faríamos com qualquer outro valor em Python. Por exemplo:

```
print(x + 7)
```

Quando o Python avalia esta instrução, ele deve calcular `x+7`. Ele se lembrará do valor que estava associado com `x`, e o acima irá mostrar `11`.

9.1) Nomes Válidos para Variáveis

Python só aceita certos nomes para suas variáveis. Nomes de variáveis em Python só podem conter letras, números, e *underscore* (`_`). Além disso, eles precisam começar com uma letra ou um *underscore*. Nomes com outros caracteres não são permitidos.

Python também tem várias `keywords` (palavras-chave) que não podem ser usadas como variáveis, já que possuem outro significado na linguagem. As palavras-chave em Python são: `False`, `None`, `True`, `and`, `as`, `assert`, `async`, `await`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `try`, `while`, `with`, `yield`.

Já encontramos algumas dessas `keywords` e encontraremos outras adiante no curso. Não se preocupe em aprendê-las agora; não há muita utilidade saber seus nomes sem saber como funcionam!

9.2) Escolhendo Nomes de Variáveis

Há muita liberdade na escolha de nomes de variáveis, mas é uma boa ideia tentar equilibrar a escolha de nomes descritivos com a escolha de nomes curtos que sejam razoavelmente fáceis de digitar.

Por exemplo, é muito difícil dizer o que o programa a seguir faz por causa de nomes de variáveis mal escolhidos:

```
a = 3.14159
b = 2

c = 2 * a * b
d = a * b ** 2

e = 4 * d
f = 4 / 3 * a * b ** 3
```

A legibilidade poderia ser melhorada drasticamente escolhendo nomes de variáveis com mais cuidado:

```
pi = 3.14159
raio = 2

circulo_circunferencia = 2 * pi * raio
circulo_area = pi * raio ** 2

esfera_area_superficie = 4 * circulo_area
esfera_volume = 4 / 3 * pi * raio ** 3
```

Claro, é possível ir longe demais na direção de nomes de variáveis descritivas. O código a seguir, que beira o absurdo, talvez seja ainda mais difícil de decifrar do que o primeiro exemplo:

```
a_proporcao_entre_a_circunferencia_e_o_raio_de_um_circulo = 6.28318
o_raio_do_formato_com_que_queremos_computar_valores = 2

a_circunferencia_de_um_circulo_com_raio_dado_pela_variavel_definida_acima
= a_proporcao_entre_a_circunferencia_e_o_raio_de_um_circulo * o_raio_do_f
ormato_com_que_queremos_computar_valores
a_area_de_um_circulo_com_raio_dado_pela_variavel_definida_acima = 1 / 2 *
a_proporcao_entre_a_circunferencia_e_o_raio_de_um_circulo * o_raio_do_for
mato_com_que_queremos_computar_valores ** 2

a_area_superficial_de_uma_esfera_com_raio_dado_pela_variavel_definida_aci
ma = 4 * a_area_de_um_circulo_com_raio_dado_pela_variavel_definida_acima
o_volume_de_uma_esfera_com_raio_dado_pela_variavel_definida_acima = 2 / 3
* a_proporcao_entre_a_circunferencia_e_o_raio_de_um_circulo * o_raio_do_f
ormato_com_que_queremos_computar_valores ** 3
```

Python executará qualquer um desses arquivos e os resultados serão os mesmos. Mas é importante escrever programas pensando se são corretos, mas também se são claros e legíveis!

10) Comentários

À medida que os programas ficam maiores e mais complicados, eles se tornam mais difíceis de ler. Linguagens de programação são densas e geralmente é difícil olhar para um pedaço de código e descobrir o que ele está fazendo ou por quê.

Por isso, é uma boa ideia adicionar notas aos seus programas para explicar em palavras o que o programa está fazendo. Essas notas são chamadas de *comentários* e começam com o símbolo `#` :

```
# calcula a porcentagem da hora decorrida
porcentagem = (minuto * 100) / 60
```

Nesse caso, o comentário aparece em uma linha por si só. Você também pode colocar comentários no final de uma linha:

```
porcentagem = (minuto * 100) / 60  # porcentagem de uma hora
```

Tudo do `#` até o final da linha é ignorado; não tem efeito na execução do programa.

Os comentários podem ser muito úteis, tanto como notas para você mesmo enquanto escreve/revisa programas e também para ajudar outras pessoas a entender seu código. Os comentários são mais úteis quando documentam partes não óbvias do código. Frequentemente, é razoável supor que o leitor pode descobrir o que o código faz; é mais útil explicar *por quê*.

Por exemplo, este comentário é redundante com o código e inútil:

```
v = 5  # atribuir 5 a v
```

Por outro lado, este comentário contém informações úteis que não estão no código:

```
v = 5  # velocidade em metros / segundo
```

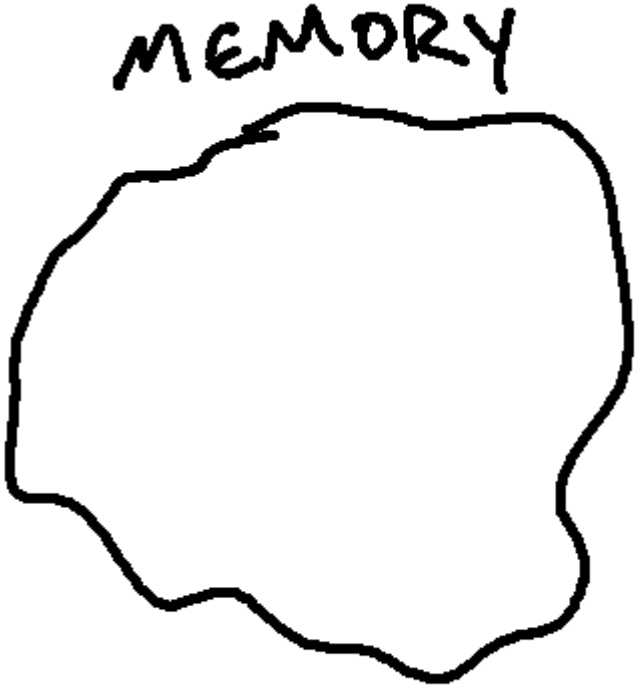
Bons nomes de variáveis podem reduzir a necessidade de comentários, mas nomes longos podem dificultar a leitura de expressões complexas. Encontrar um equilíbrio é importante.

11) Diagramas de Ambiente

Anteriormente, começamos a construir um modelo mental de como o Python se comporta, desenvolvendo o modelo de substituição para avaliação de expressão. Agora, com variáveis, teremos que expandir nosso modelo.

Nesta seção, falaremos sobre como o Python avaliaria um programa curto, usando um tipo específico de desenho (chamado de *diagrama de ambiente*) para ajudar a acompanhar as coisas.

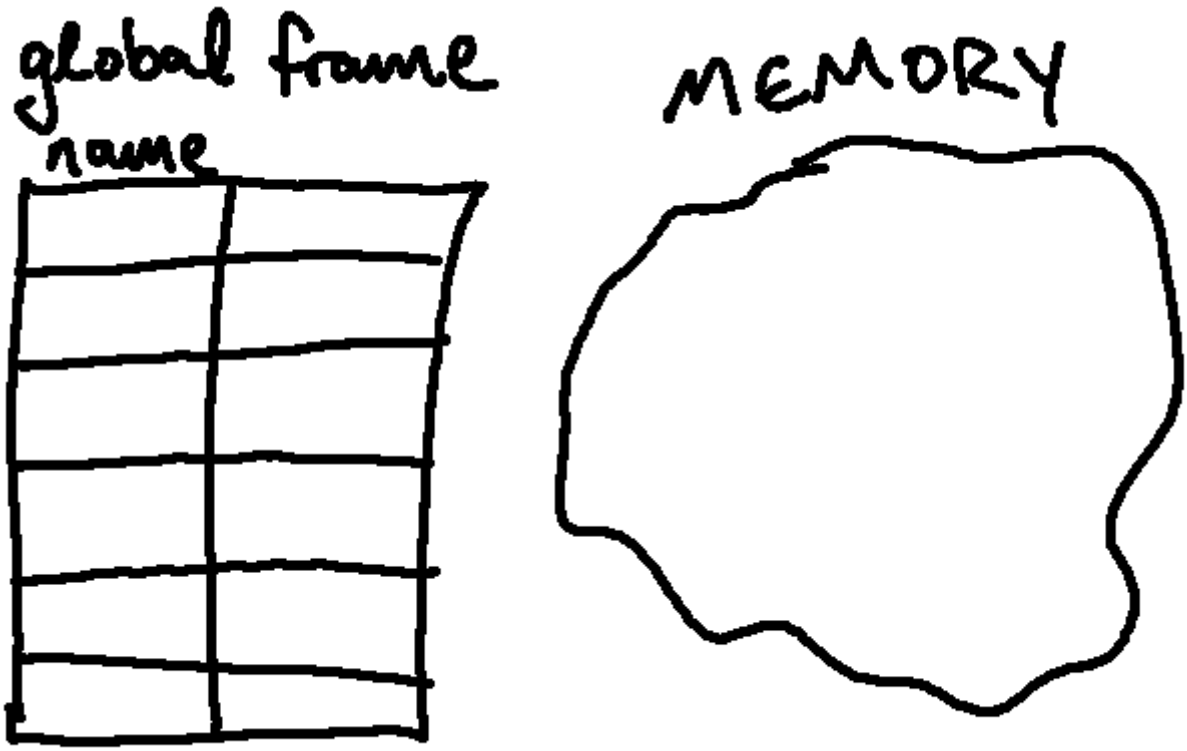
Além de ser capaz de avaliar expressões, Python também possui certa *memória*. Por enquanto, pensaremos na memória como um lugar abstrato onde o Python armazena/rastreia os objetos. Nos desenhos, a memória é geralmente representada como uma bolha ou ameba:



Quando Python precisa se lembrar de um objeto, esse objeto é armazenado na memória. Além disso, Python precisa de uma maneira de associar nomes aos objetos armazenados. Ele faz isso por meio de uma construção chamada *frame* ("quadro"), em que podemos pensar como uma tabela de pesquisa: dado um nome, ele nos direciona para o objeto associado na memória.

Por enquanto, nosso modelo consistirá de exatamente um *frame*, ao qual nos referimos como *Frame* Global ou Quadro Global. Com as ferramentas que atualmente temos disponíveis, é aqui que os mapeamentos de nome a objeto serão mantidos. No futuro veremos situações em que podemos ter mais de um *frame*.

Nosso ponto de partida para um diagrama de ambiente conterá tanto essa ideia abstrata de memória e o Quadro Global. Por enquanto, pensaremos no Global Frame como estando completamente vazio quando o Python iniciar. Portanto, quando o Python é iniciado pela primeira vez, nosso modelo atual (na forma de um diagrama de ambiente) se parece com isto:

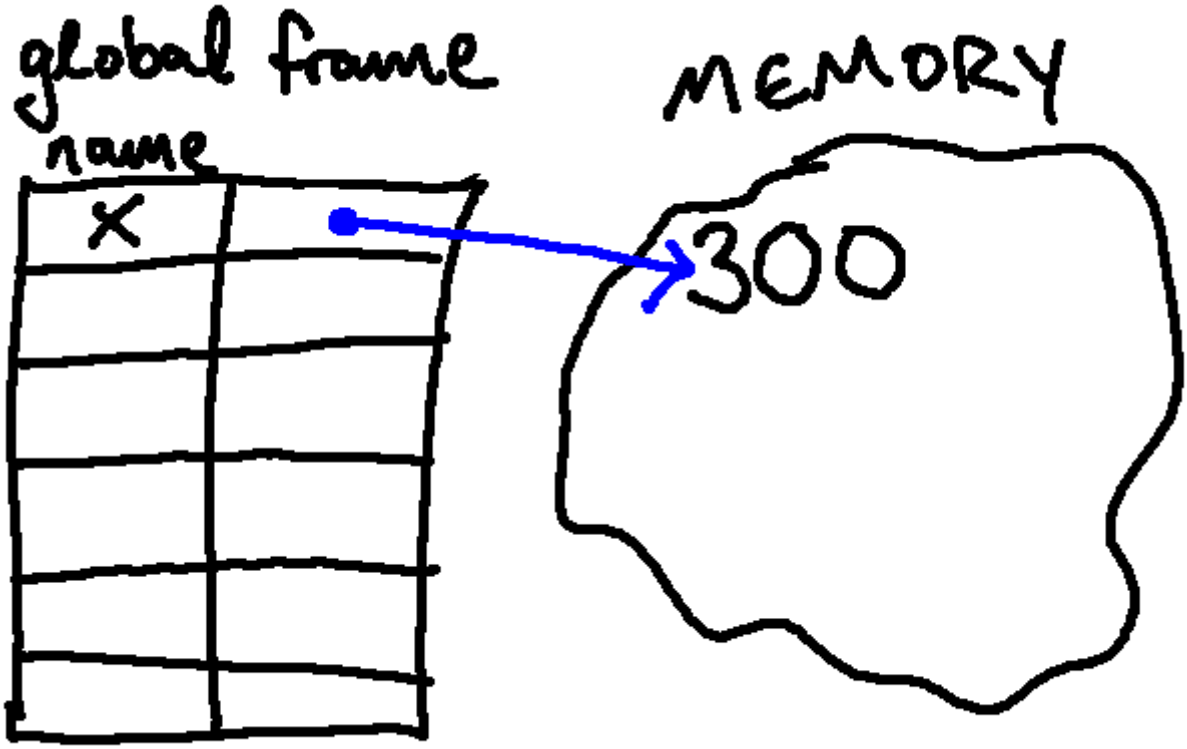


Conforme novas atribuições são feitas, Python armazenará os objetos relevantes na memória e fará novas associações no Global Frame. Como exemplo, vamos considerar o seguinte programa:

```
x = 300
print(x + 2)
y = 2 + 5
z = x + y
y = x
print(z)
```

Lembre-se de que Python avaliará esse programa uma linha de cada vez. Ele começa avaliando a primeira linha, `x = 300`. Quando o Python avalia esta linha, ele armazena o resultado da expressão à direita de `=` na memória (neste caso, `300`) e associa o nome `x` a esse valor. Em nossos diagramas de ambiente, podemos denotar essa relação colocando `300` na memória, fazendo uma entrada no Quadro Global para `x` e desenhando uma seta de `x` para o novo valor.

Então, depois de executar essa linha, nosso diagrama de ambiente agora se parece com isto (aqui, a seta está em azul apenas para que se destaque um pouco):

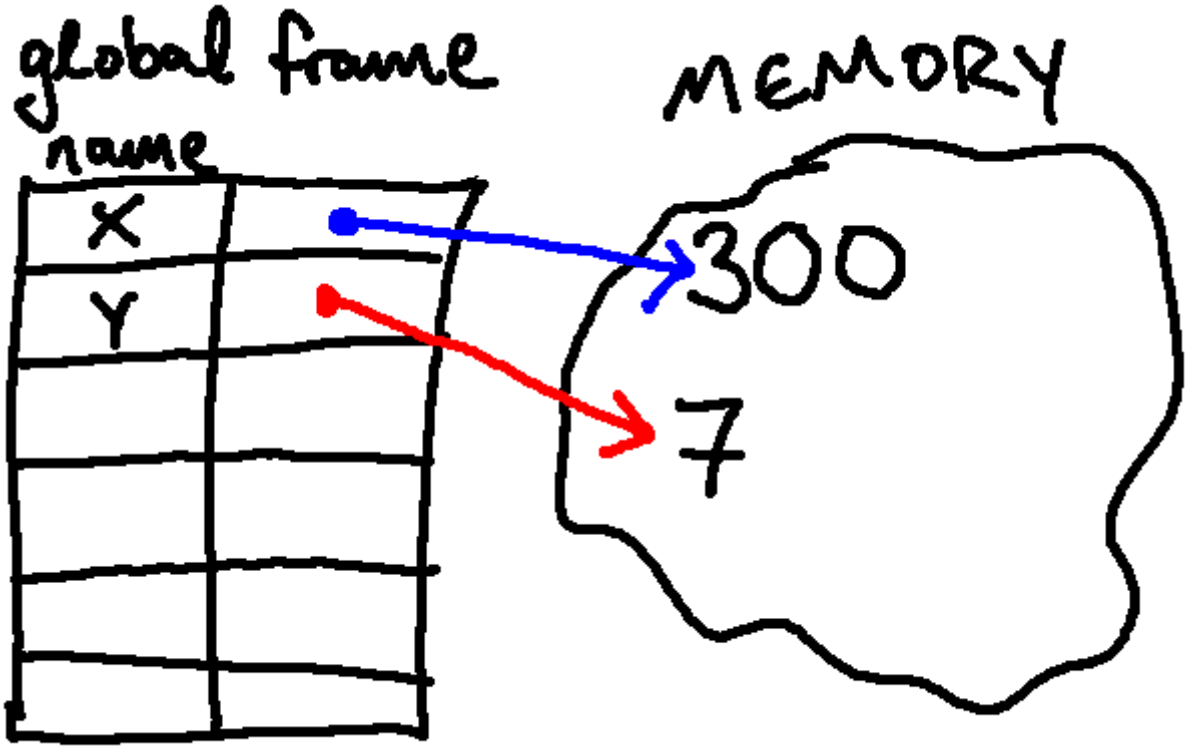


Em seguida, executamos a próxima linha, `print(x + 2)`. Ao executar esta linha, Python deve avaliar `x + 2`. Para dar conta disso, precisamos que nosso modelo de substituição seja capaz de considerar as variáveis. Simplificando, quando o Python chega a um ponto em que precisa avaliar uma variável, ele a procura no quadro global e a substitui no valor associado.

Nesse caso, temos:

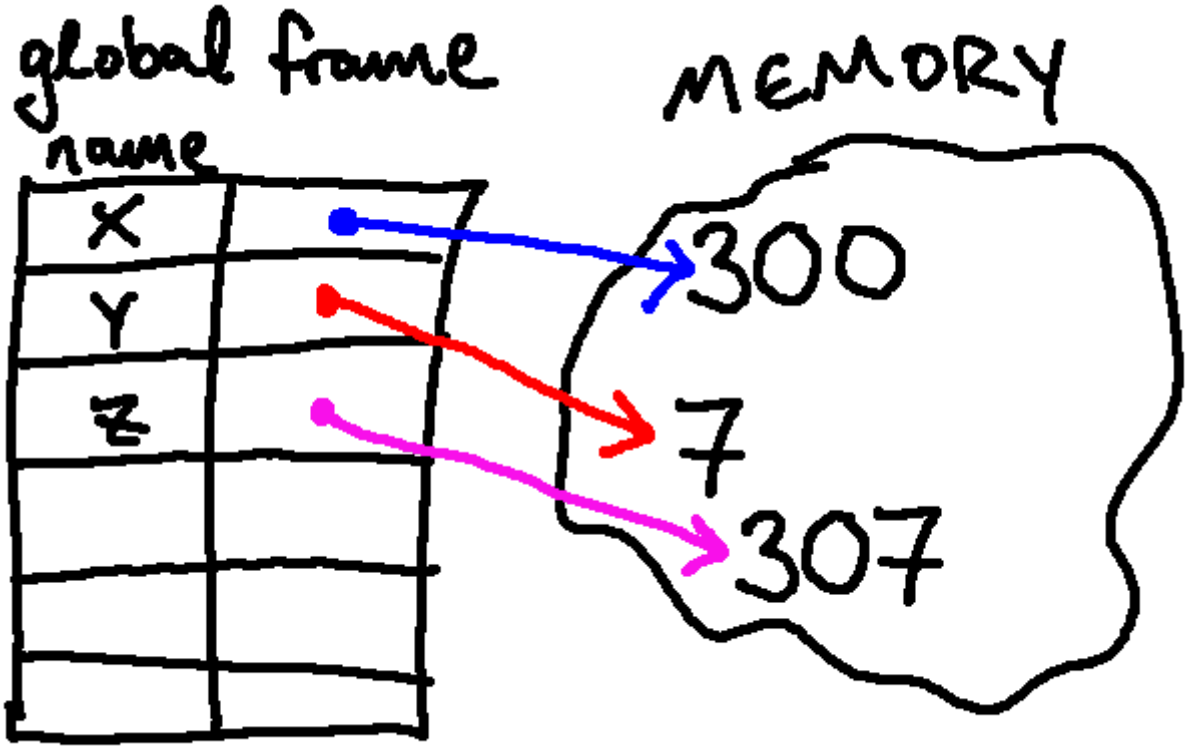
- `x + 2` (procura o valor de `x`)
- `300 + 2` (calcula a adição)
- `302` então `302` será exibido na tela.

Em seguida, Python executará a linha: `y = 2 + 5`. Mais uma vez, Python avaliará a expressão à direita do sinal de igual (`2 + 5`), armazenará o resultado na memória como `7` e associará o nome `y` a esse objeto:



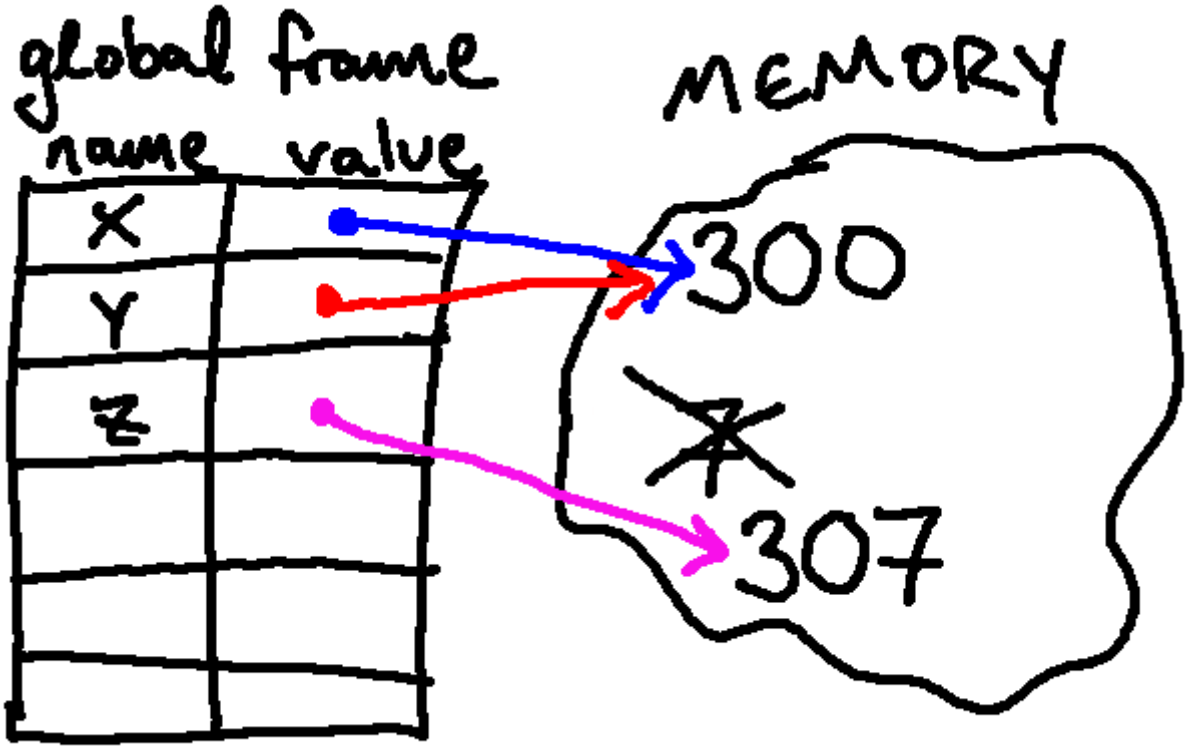
Então Python executará: `z = x + y`. A expressão à direita de `=` é avaliada. Usando o modelo de substituição:

- `x + y`
- `300 + y`
- `300 + 7`
- `307` O valor `307` é adicionado à memória e o nome `z` é associado ao valor `307`:



Em seguida, avaliamos `y = x`. É importante ter cuidado aqui. Observe que o Python começa avaliando a expressão no lado direito do sinal de "igual". Avaliar `x` procura o nome `x` e encontra o valor `300` armazenado na memória.

Em seguida, Python associa o nome `y` a esse valor. Isso altera o valor associado a `y`, de modo que `y` agora está associado ao resultado da avaliação de `x` (`300`). Graficamente, isso resulta no seguinte:



Algumas observações do diagrama atualizado:

1. O valor `y` agora aponta para *o exato mesmo objeto* que está associado ao nome `x` (porque encontramos esse objeto procurando `x` e seguindo sua seta até um objeto na memória). Se tivéssemos escrito `y = 300`, faríamos um novo objeto com o mesmo tipo e valor de nosso `300` original e y apontaria para isso (e, portanto, nesse caso, teríamos *dois objetos* `300` *diferentes* na memória, com `x` apontando para um deles e `y` apontando para o outro).
2. Depois que a definição de `y` for atualizada, não haverá mais nomes associados ao `7` na memória. Normalmente, quando há um valor na memória sem variáveis fazendo referência a ele, Python o remove da memória. Esse processo é chamado de *coleta de lixo* porque envolve a limpeza dos objetos que sobraram na memória que não estão mais sendo usados. Em nossos diagramas de ambiente, vamos riscar objetos que foram "coletados".

Finalmente, ao executar `print(z)`, Python procura `z`, que ainda está associado ao valor `307` (apesar do fato de que a associação de `y` mudou), então `307` é exibido.

Tente agora:

Considere o programa em Python:

```
x = 4
x = x + 1
print(x)
```

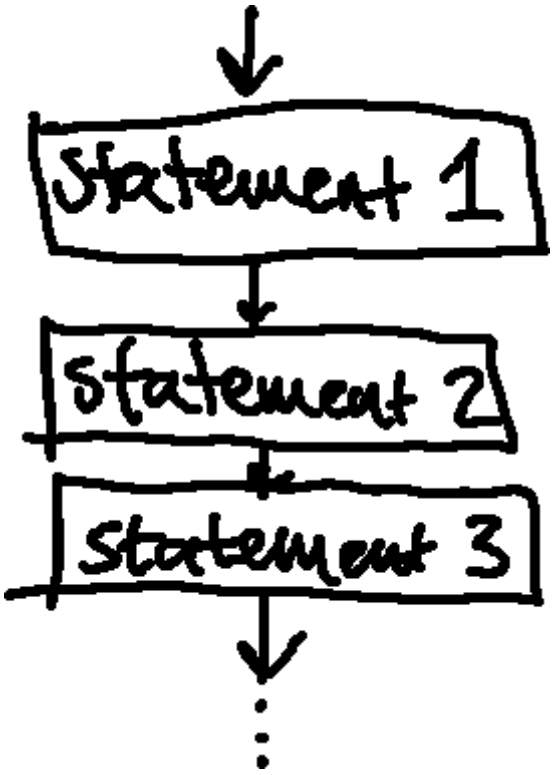
Desenhe um diagrama de ambiente que represente a evolução desse programa e use-o para predizer qual valor será exibido quando executarmos o programa. Então execute o programa. O resultado é o que você esperava?

▶ Mostrar/Esconder

---

12) Execução Condicional

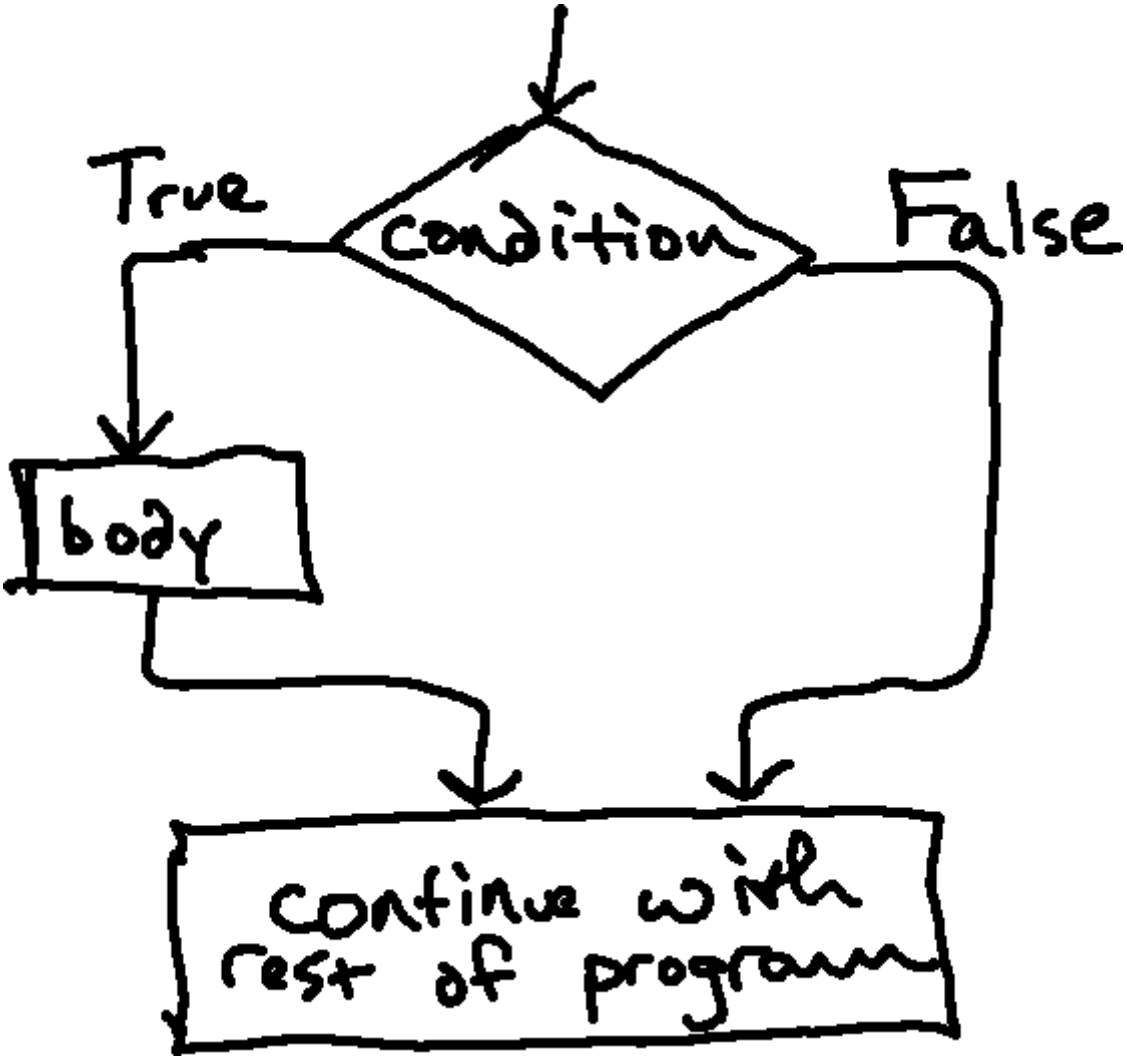
Até agora, todos os nossos programas continuaram de uma maneira relativamente direta: Python executou todas as instruções em um programa exatamente na ordem em que foram especificadas.



Para escrever programas úteis, no entanto, quase sempre precisamos da capacidade de checar condições e alterar o comportamento do programa de acordo. *Declarações condicionais* nos dão essa capacidade. A forma mais simples é a instrução `if`:

```
# valor absoluto
if x < 0:
    x = -x
```

A expressão booleana após `if` é chamada de *condição*. Se for verdadeira (ou seja, se for avaliada como `True`), a instrução indentada (o *corpo*) será executada. Se não, nada acontece. Essa estrutura é representada pelo fluxograma:



Note que também podemos ter várias instruções no corpo; todas as instruções recuadas no mesmo nível fazem parte do corpo e são executadas se a condição for verdadeira. Por exemplo:

```
# Este exemplo também apresenta um novo conceito:
# podemos exibir uma sequência de caracteres na
# tela literalmente, colocando-os entre aspas.

print("Isso sempre será exibido")
if x < 0:
    print("x é negativo")
    print("então exibiremos")
    print("algumas coisas extras")
print("Isso sempre será exibido também")
print("E isso também!")
```

Nesse caso, se `x` for menor que `0`, Python executará todas as três linhas indentadas; caso contrário, ele ignorará todas elas. Observe que apenas a execução das linhas *indentadas* é afetada pela condição; as duas últimas linhas não fazem parte da condicional e são sempre executadas.

Não há limite para o número de afirmações que podem aparecer no corpo, mas deve haver pelo menos uma (levantará um erro se não houver uma). Ocasionalmente, é útil ter um corpo sem instruções (geralmente "segurando espaço" para código que você ainda não escreveu). Nesse caso, você pode usar a instrução `pass`, que não faz nada.

```
if x < 0:
    pass # não faça nada
```

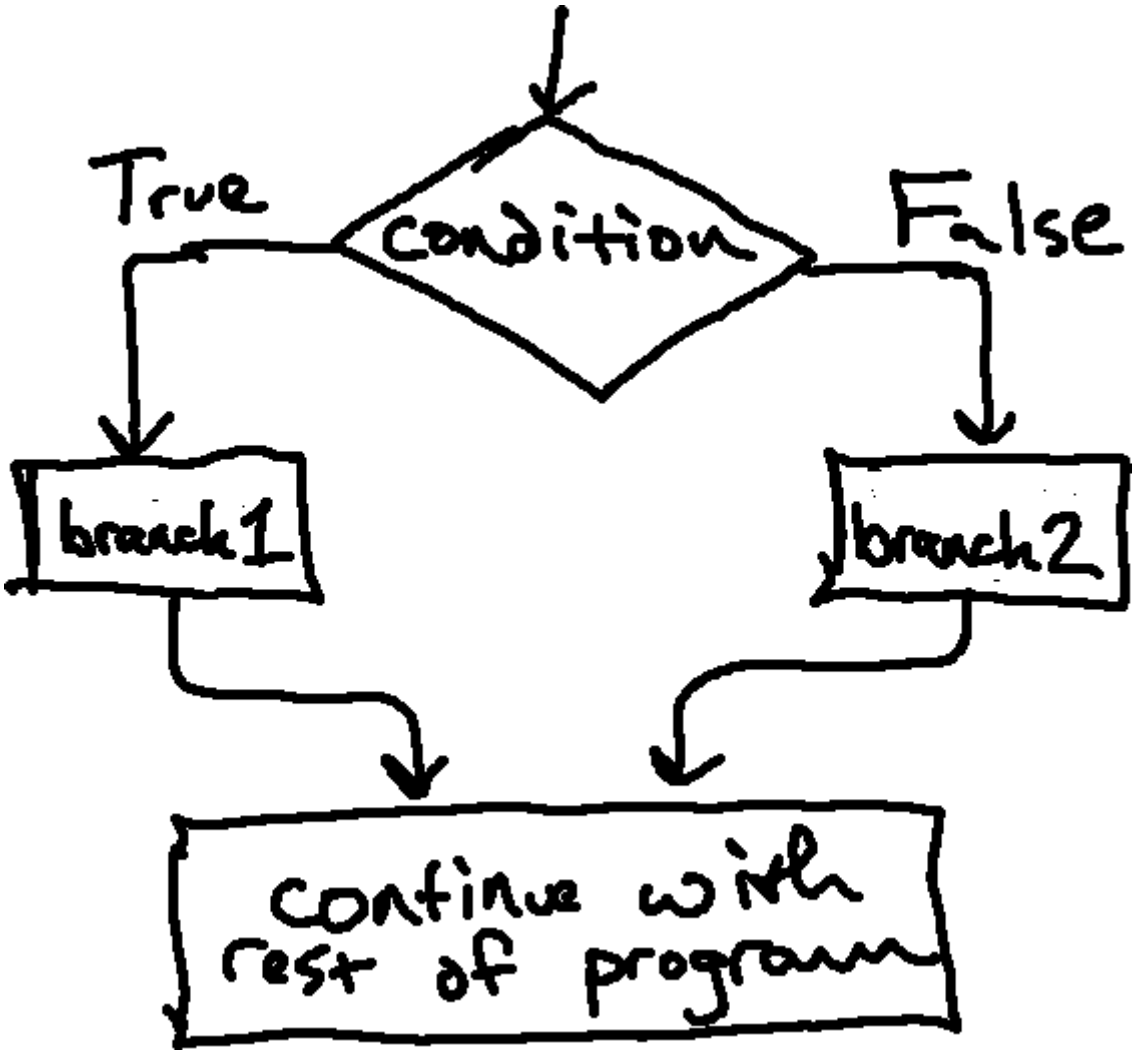
As instruções `if` têm uma estrutura interessante: um cabeçalho (*header*) seguido por um corpo indentado. Declarações como essa são chamadas de *declarações compostas*. Aprenderemos mais sobre mais alguns tipos de instruções compostas no restante do curso.

Uma segunda forma da instrução `if` faz com que o Python execute uma das várias alternativas possíveis de código. Nessa estrutura, existem duas possibilidades e a condição determina qual delas é executada. A estrutura tem o formato:

```
if x % 2 == 0:
    print("x é par")
else:
    print("x é ímpar")
```

Se o resto quando `x` é dividido por 2 for 0, então sabemos que `x` é par e exibiremos uma mensagem nesse sentido. Se a condição for falsa, o segundo conjunto de instruções será executado. Como a condição deve ser verdadeira ou falsa, exatamente uma das alternativas será executada. As alternativas são chamadas de *ramos*, porque são ramos no fluxo de execução.

Essa estrutura (que poderia ser chamada de "execução alternativa") é representada pelo seguinte fluxograma:



12.1) Condicionais Encadeados

Às vezes, existem mais de duas possibilidades e precisamos de mais de dois ramos. Uma maneira de expressar uma computação como essa é uma *condicional encadeada*:

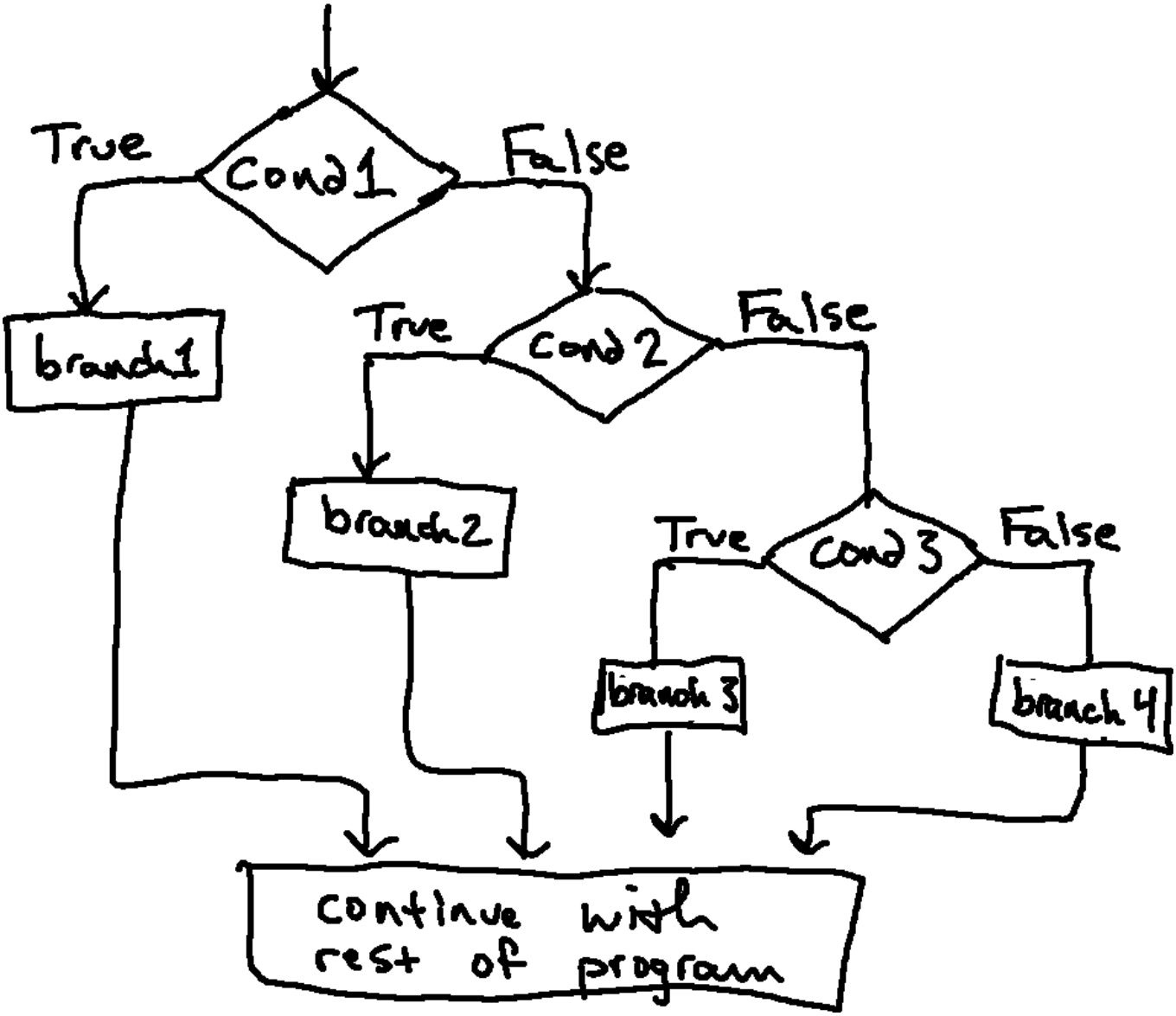
```
if x < y:
    print('x é menor que y')
elif x > y:
    print('x é maior que y')
else:
    print('x e y são iguais')
```

`elif` é uma abreviatura de "else if" ("senão se"). Novamente, exatamente um ramo será executado. Não há limite para o número de declarações `elif` . Se houver uma cláusula `else` , ela deve estar no final, mas não é necessário que haja uma.

```
if escolha == 'a':
    print('A escolha foi um')
elif escolha == 'b':
    print('A escolha era b')
elif escolha == 'c':
    print('A escolha foi c')
else:
    print('A escolha foi outra ...')
```

Cada condição é checada em ordem. Se o primeiro for falso, o próximo será verificado e assim por diante. Se um deles for verdadeiro, o ramo correspondente é executado e a instrução termina. Mesmo se mais de uma condição for verdadeira, apenas o primeiro ramo verdadeiro será executado.

A estrutura do código acima é mostrada no seguinte fluxograma:



12.2) Condicionais "Aninhados" (*Nested*)

As ramificações de uma condicional podem conter *código Python arbitrário* (pode ser qualquer código). Isso significa que, entre outras coisas, uma condicional também pode ser aninhada em outra ("uma dentro da outra"). Poderíamos ter escrito o exemplo acima como:

```
if x == y:
    print("x e y são iguais")
else:
    if x < y:
        print("x é menor que y")
    else:
        print("x é maior que y")
```

A condicional externa contém dois ramos. O primeiro ramo contém uma declaração simples. A segunda ramificação contém outra instrução `if` , que possui duas ramificações próprias. Essas duas ramificações são declarações simples, embora também pudessem ser declarações condicionais.

13) Bugs

Infelizmente, como a maioria dos outros tipos de pessoas, programadores ocasionalmente cometem erros. Isso é verdade em todos os níveis de experiência e até mesmo para programadores que estão sendo muito cuidadosos para evitar erros.

Por razões históricas interessantes, erros em programas de computador são chamados de *bugs* ("insetos"), e o ato de corrigir esses erros é chamado *debugging*.

Programação, e especialmente *debugging*, às vezes traz à tona emoções fortes. Se você está lutando com um bug difícil, pode ficar com raiva, desanimado ou envergonhado.

Há evidências de que as pessoas naturalmente respondem aos computadores como se fossem pessoas. Quando funcionam bem, pensamos neles como companheiros de equipe e, quando são obstinados ou rudes, respondemos a eles da mesma forma que respondemos a pessoas rudes e obstinadas.

Mas é importante ter em mente como os computadores são diferentes de pessoas (o que pode ser a fonte de muitos bugs!). Apesar de incrível velocidade e precisão, eles têm uma total falta de empatia e uma incapacidade de compreender o "quadro geral" do que nós, como programadores, estamos tentando atingir.

Cada peça desse quebra-cabeça tem limitações. Humanos são bons em pensar criativamente, mas somos muito lentos e temos problemas para lembrar das coisas. Por outro lado, Python é ótimo em lembrar de coisas e trabalhar rapidamente, mas não é tão criativo ou inteligente quanto os humanos podem ser. Os pontos fortes de uma peça são os pontos fracos da outra, e vice-versa, e é isso que pode fazer essa união funcionar tão bem! Juntos, no melhor dos casos, o ser humano e o Python formam um superorganismo que é bom em todas essas coisas!

Mas ainda há um problema na interface: para fazer o computador trabalhar para nós, temos que converter nossos pensamentos em programas, que o Python então interpreta; e esse processo pode estar sujeito a (muitos) erros. Nesta seção, falaremos brevemente sobre alguns tipos de bugs que podem aparecer em seus programas e por quê. Ao longo do curso, trabalharemos para entender como localizar e eliminar esses bugs.

13.1) Perdido na Tradução

Cabe a nós traduzir nosso plano para uma linguagem formal que o computador (em particular, o interpretador Python) possa entender. Mas a rigidez da maioria das linguagens de programação apresenta uma barreira, em parte porque são muito diferentes das linguagens que falamos na vida cotidiana.

Línguas naturais (as línguas que as pessoas falam, como inglês ou português, que evoluem naturalmente) são robustas contra vários tipos de erros ou ambiguidades, em parte porque são interpretadas por humanos. Por exemplo, em português, se ue escerver uma plavra ou daus errdas, voc aind conseguue entener. Ou se cometermos erros gramaticais, muitas vezes ainda conseguimos entender o que queremos dizer.

Ao interpretar sentenças em uma linguagem natural, geralmente usamos pistas contextuais, conhecimento prévio e redundância para lidar com coisas como ambiguidade.

Enquanto as línguas naturais são geralmente relativamente livres na forma e cheias de ambiguidade, *linguagens formais* (linguagens projetadas por humanos para aplicações específicas) tendem a ser muito rígidas na estrutura e literais em significado. Python é um exemplo de linguagem formal projetada para expressar computações. Infelizmente, o computador não pode usar pistas contextuais ou conhecimento prévio para ajudar a lidar com imprecisões ou ambiguidades nos programas que lhe damos. Portanto, temos que traduzir nossos planos para Python com muito cuidado.

Como a maior parte de nossa experiência envolve o trabalho com linguagens naturais, pode ser difícil traduzir ideias para uma linguagem formal, o que pode resultar em erros em nossos programas. Já vimos dois tipos de erros que podem resultar como parte desse processo de tradução: erros de sintaxe e erros de tempo de execução.

- **Erros de sintaxe:** "Sintaxe" refere-se à estrutura de um programa e às regras sobre essa estrutura. Por exemplo, os parênteses devem vir em pares correspondentes, então `(1 + 2)` é sintaticamente válido, mas `8)` não é; é um *erro de sintaxe*. Python é muito específico quanto à estrutura sintática dos programas que avalia.

  Se houver um erro de sintaxe em qualquer lugar do programa, Python exibirá uma mensagem de erro e será encerrado, e você não conseguirá executar o programa. Durante as primeiras semanas de sua carreira de programação, você pode gastar muito tempo rastreando erros de sintaxe. Conforme você ganha experiência, cometerá menos erros de sintaxe e os encontrará mais rápido.
- **Erros de tempo de execução:** O segundo tipo de erro é um erro de tempo de execução, assim chamado porque o erro não aparece até que o programa tenha iniciado a execução. Esses erros também são chamados de *exceções* porque geralmente indicam que algo excepcional (e ruim) aconteceu.

  Esse tipo de erro resulta em muitos casos quando um programa sintaticamente válido tem outros problemas. Por exemplo, a expressão `x + y` é sempre sintaticamente válida, mas se `x` tiver o valor `None` e `y` tiver o valor `6` quando a expressão for calculada, Python encerrará com um `TypeError`, informando que os tipos dos dois operandos eram incompatíveis com a operação em questão.

  Outro exemplo seria tentar, por exemplo, `print(alguma_variavel)` se `alguma_variavel` ainda não tivesse sido definida. Não há nada sintaticamente errado com essa expressão e, portanto, o Python não notará até que tente ser executado. Mas se tentássemos executar esse código, Python seria encerrado em uma tempestade de texto em vermelho (desta vez, com um `NameError` porque não foi possível encontrar o nome em questão no quadro global).

  Os erros de tempo de execução são um pouco mais raros nos programas simples que você verá nas primeiras atribuições, portanto, pode demorar um pouco até que você encontre um.

13.2) Python dá Trabalho

Em certo sentido, há algo bom sobre os dois tipos de erros na seção anterior: Python reclama muito para ambos os tipos de erros. Ele para de executar qualquer código e relata uma mensagem de erro.

Mas há ainda outro tipo de erro que ainda não foi discutido e é muito mais sutil.

Tente agora:

Por exemplo, considere o seguinte programa projetado para calcular e imprimir o valor absoluto de `x`:

```
if x > 0:
    print(-x)
else:
    print(x)
```

Você consegue identificar o erro neste programa?

▶ Mostrar/Esconder

Esse tipo de erro é conhecido como *erro semântico* ("semântico" quer dizer: relacionado a significado). Se houver um erro semântico em seu programa, ele será executado sem gerar mensagens de erro, mas não fará a coisa certa. Vai fazer outra coisa. Especificamente, ele fará exatamente o que você disse para fazer.

Como esses tipos de erros são tão traiçoeiros, tomaremos muito cuidado ao longo deste curso (especialmente depois de desenvolvermos mais ferramentas nas seções posteriores) para testar partes individuais de nossos programas para ter certeza de que estão funcionando antes de juntá-los.

14) Resumo

Nesta primeira leitura, cobrimos bastante terreno. Mais importante, começamos a construir um modelo de como Python responde a vários tipos de comandos ou instruções que podemos dar a ele. Aprendemos sobre vários tipos primitivos do Python, sobre maneiras de combiná-los em formas mais complicadas e sobre como armazenar os resultados de nossos cálculos para que possamos reutilizá-los.

Ao longo do caminho, cutucamos um pouco o Python, testando coisas diferentes para descobrir como ele se comporta.

Neste momento, nosso modelo mental de Python inclui várias peças:

- Uma noção de vários tipos de objetos Python (`int` , `float` , ...)
- Uma série de operadores para combinar esses objetos (`+` , `<` , `and` , ...)
- Um modelo de como Python avalia expressões (o modelo de substituição)
- Um modelo de como o Python armazena objetos na memória (diagramas de ambiente)
- Um meio de controlar o fluxo de programas (execução condicional)

Neste conjunto de exercícios, você obterá alguma prática com todas essas ferramentas. No próximo conjunto de leituras e exercícios, vamos expandir esses modelos, introduzindo vários novos tipos de objetos Python e algumas novas ferramentas para controlar o fluxo de programas.