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 ¶

Até agora, aprendemos sobre uma série de ferramentas úteis em Python. Aprendemos sobre:

- vários tipos de *objetos* Python que nos permitem representar uma variedade de coisas na memória do Python (os tipos que vimos até agora incluem números, strings, booleanos, o valor especial `None` e objetos compostos como listas e tuplas)
- vários *mecanismos de controle de fluxo* que nos permitem controlar a ordem em que as instruções em um programa Python são executadas (os mecanismos de fluxo de controle que vimos até agora incluem loops `while` e `for`). Essas são ferramentas realmente poderosas e, como você viu através dos exercícios, são suficientes para realizar uma ampla variedade de coisas! No entanto, nesta seção, aprenderemos sobre um meio de abstração incrivelmente poderoso que ajudará você a gerenciar complexidade à medida que os programas que escreve se tornam cada vez mais complicados: *funções*.

Vamos começar nossa discussão sobre funções considerando o seguinte trecho de código, que é projetado para calcular o resultado da avaliação de um polinômio (representado como uma lista de coeficientes) em um determinado valor numérico:

```
coeffs = [7, 9, 2, 3]
x = 4.2

result = 0
for index in range(len(coeffs)):
   result = result + coeffs[index] * x ** index
print(result)
```

Este é um bom pedaço de código no contexto de um programa que requer a avaliação de um polinômio, mas em certo sentido não é tão útil quanto poderia ser, pois funciona apenas para os valores de `coeffs` e `x` dados acima.

É possível, no entanto, imaginar um programa maior que requer a avaliação de *vários* polinômios. Com as ferramentas que temos disponíveis até agora, se quiséssemos usar o código acima para esse fim, teríamos que copiar o código várias vezes e colá-lo em novos locais. Dependendo do que nosso programa está fazendo, podemos precisar alterar os nomes das variáveis `coeffs` , `x` e `result` para evitar que eles sobrescrevam os valores que calculamos para outros polinômios. Isso é uma encheção de saco! Além disso, se encontrarmos um bug em nossa implementação, teríamos que voltar e corrigi-lo em cada cópia que fizermos desse trecho de código!

Seria ótimo ser capaz de generalizar a noção desse cálculo para que pudéssemos executá-lo em entradas arbitrárias como parte de um programa maior. Acontece que um novo tipo de objeto Python, chamado de *função*, nos permitirá fazer isso!

Funções são indiscutivelmente a ferramenta mais poderosa que qualquer programador pode ter em seu kit de ferramentas, mas pode ser um pouco complicado entender como o Python as manipula. Como tal, vamos apresentar relativamente poucos tópicos novos nesta seção, para que possamos nos concentrar nas funções em particular, em como o Python avalia o código dentro de uma função e em como elas podem ser usados para aumentar a modularidade dos programas que escrevemos.

1) Funções

Uma *função* é um tipo de objeto Python que representa uma computação abstrata. Talvez ajude pensar em uma função como um pequeno programa em si mesmo, que executa uma tarefa específica. Internamente, isso é o que uma função realmente *é*: é uma sequência generalizada de instruções que o Python pode avaliar para calcular um resultado. Essa talvez não seja a definição mais elegante do mundo, então vamos prosseguir com um exemplo.

Python vem com várias funções integradas (*built-in*) e, de fato, já vimos vários exemplos de funções em Python. Por exemplo, já aprendemos sobre `len`, que calcula o comprimento de uma sequência de entrada. Poderíamos usar `len` dentro de um programa, por exemplo, avaliando a seguinte expressão: `len("futebol")`

Neste exemplo, o nome da função com a qual estamos trabalhando é `len`. Os parênteses indicam que queremos "chamar" a função (também chamado de "invocar" a função), que significa avaliar a sequência de declarações que ela representa. A expressão entre parênteses (aqui, `"futebol"`) é chamada de *argumento* da função. Nesse caso, o resultado é um número inteiro que representa o comprimento do argumento.

É comum dizer que uma função "recebe" um ou mais argumentos como entrada e "retorna" um resultado. Esse resultado também é chamado de *valor de retorno*. Nesse caso, `len` é um objeto `function` e o resultado da chamada é um `int`. Mas as funções podem retornar valores de *qualquer* tipo (até outras funções como veremos no próximo set de leituras)!

Podemos tratar o resultado da chamada de uma função da mesma forma que trataríamos qualquer outro objeto Python. Por exemplo, poderíamos:

```
print(len("futebol")) # imprime o resultado
x = len("bola") # armazena o resultado em uma variável
y = len("menino ney") + 27/2 # combina o resultado com outras operações
```

1.1) Argumentos múltiplos

Algumas funções recebem mais de um argumento. Para especificar isso, separamos os argumentos com vírgulas dentro dos parênteses associados a uma chamada de função. Por exemplo, considere a função `divmod` integrada do Python, que usa dois argumentos:

```
print (divmod (19, 4)) # divmod retorna uma tupla; isso irá imprimir (4,
 3)
```

Tente agora:

Isso é uma espécie de aparte, mas vale a pena mencionar que o Python fornece documentação para todos os seus recursos integrados (built-ins), o que pode ser muito útil para determinar como usar as funções do Python. Por exemplo, consulte a seção sobre `divmod` (https://docs.python.org/3/library/functions.html#divmod).

1.1.1) Formas alternativas de funções anteriores

Acontece que duas das funções com as quais já tratamos têm formas alternativas que possuem mais de um argumento, o que pode ser útil em seus programas futuros:

- Se `print` receber mais de um argumento, ele imprimirá todos os seus argumentos na mesma linha, separados por espaços. Se não for fornecido nenhum argumento (ou seja, `print()`), ele simplesmente fará uma linha em branco.
- `range` tem três formas:
  - Se `range` for dado um único inteiro x, o objeto que ele retorna contém os números de 0 a $x - 1$, inclusive.
  - Se `range` for fornecido com dois inteiros x e y, o objeto que ele retorna contém os números de x a $y - 1$, inclusive.
  - Se `range` for dado três inteiros x, y e z, o objeto que ele retorna contém todos os valores $x + i \times z$ de modo que $x \leq x + i \times z < y$, em ordem crescente de i.

Coloquialmente, os três argumentos para `range` são frequentemente referidos como "start", "stop" e "step": o primeiro é o valor para começar (inclusivo), o segundo é o valor em que parar (exclusivo) e o terceiro é quantos números pular cada vez (tamanho do "passo").

Tente agora:

Experimente com essas várias formas para ter uma ideia de como elas se comportam. Tente imprimir vários valores em uma única linha. Tente imprimir vários intervalos. Já que `range` não retorna uma lista (mas, em vez disso, um objeto `range` especial), você precisa converter esse objeto em uma lista ou tupla para ver os objetos dentro dela (por exemplo, `list(range(9))` ou `tuple(range(1, 4))`).

Tente agora:

Como vimos acima, `print` também é uma função! Qual é o valor de retorno de `print`? Tente armazenar o resultado de uma chamada de `print` em uma variável e, em seguida, exibi-lo (novamente com `print`!).

▶ Mostrar/Esconder

O que o código a seguir exibirá?

```
print(print(print(10)))
```

▶ Mostrar/Esconder

1.2) Modelo de Substituição

Antes de irmos muito mais longe, precisamos pensar sobre como o Python avalia as funções em nosso modelo de substituição. Para avaliar uma chamada de função, o Python executa as seguintes etapas:

- Procura o valor à esquerda dos parênteses
- Avalia cada um dos argumentos da esquerda para a direita
- Chama a função com os resultados da avaliação dos argumentos

---

Tente agora:

Use o modelo de substituição para prever o resultado da avaliação da seguinte expressão:

```
divmod (17 + 2.0, len ("ca" + "ts"))
```

▶ Mostrar/Esconder

Tente agora:

O que acontece quando você tenta executar o seguinte trecho de código? Por que isso aconteceu?

```
x = 2
z = x(3.0 + 4.0)
print(z)
```

▶ Mostrar/Esconder

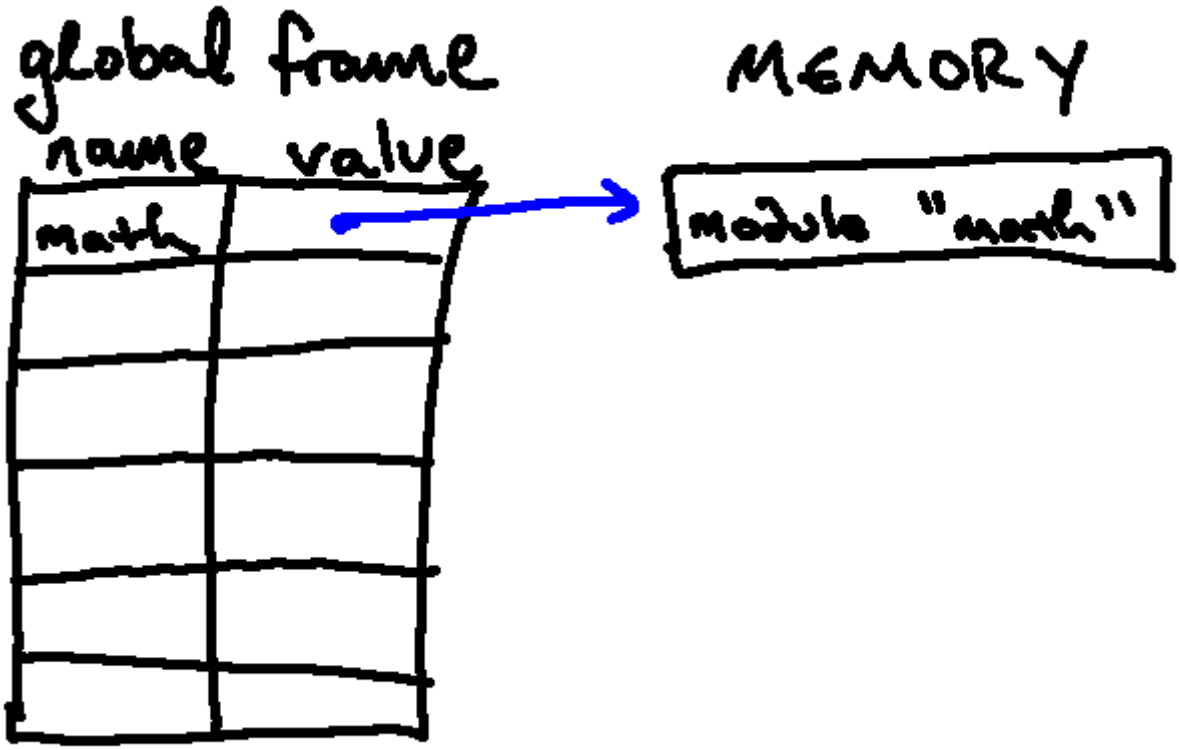
2) Importações e Notação de Ponto

Python também fornece uma grande biblioteca de outras funções e constantes que não estão disponíveis por padrão, mas que podem ser *importadas* e usadas em seu código. Esses objetos estão disponíveis em coleções chamadas *módulos*.

Um uso comum de *importações* é obter acesso a funções e constantes definidas em um módulo Python chamado `math`. Antes de podermos usar os objetos em um módulo, precisamos importá-los com uma instrução `import`, por exemplo:

```
import math
```

Esta declaração cria um *objeto de módulo* na memória e o associa ao nome `math`. Em nossos diagramas de ambiente, vamos denotar isso como:



Podemos acessar funções ou constantes de dentro do módulo usando a *notação de ponto*. Por exemplo, para pesquisar e imprimir a constante associada ao número e, poderíamos fazer o seguinte (depois de usar `import` como acima):

```
print(math.e)
```

Para avaliar uma expressão como essa, Python primeiro procura o nome `math`, encontrando o módulo armazenado na memória. O "ponto" (`.`) Então diz ao Python para procurar *dentro* desse módulo por algo com o nome `e`. Nesse caso, como o módulo `math` realmente contém algo chamado `e`, ele encontra esse objeto; então veríamos:

```
2.718281828459045
```

O módulo `math` contém outras constantes (`tau`, `pi` e outras), mas também contém funções, como `sin` e `log`. Podemos pesquisar esses objetos usando a notação de ponto também e, em seguida, invocá-los como faríamos com outras funções; por exemplo:

```
print(math.sin(3 * math.pi / 4))
```

Tente agora:

Python fornece documentação útil para todos os módulos disponíveis em uma instalação base do Python. Tente fazer uma pesquisa na web por "Python Math Module" e encontre o resultado que está associado ao Python 3 (não ao Python 2!). Se a sua versão exata do Python não estiver disponível nos resultados da pesquisa na web, você pode ir para a página do Python 3 e usar o menu suspenso no canto superior esquerdo para escolher uma versão mais próxima da que você está executando.

Tente agora:

Só por diversão, vamos calcular $\log_3 82$.

É sempre bom saber o que esperar de um programa de trabalho. Antes de executar seu código, tente adivinhar (aproximadamente) qual valor você espera ver. *Dica*: 82 é bem próximo de 81; você pode usar isso para chegar a uma estimativa razoável para o valor em questão? Se obtivéssemos algo drasticamente diferente disso, provavelmente cometemos um erro ao inserir o programa em Python!

▶ Mostrar/Esconder

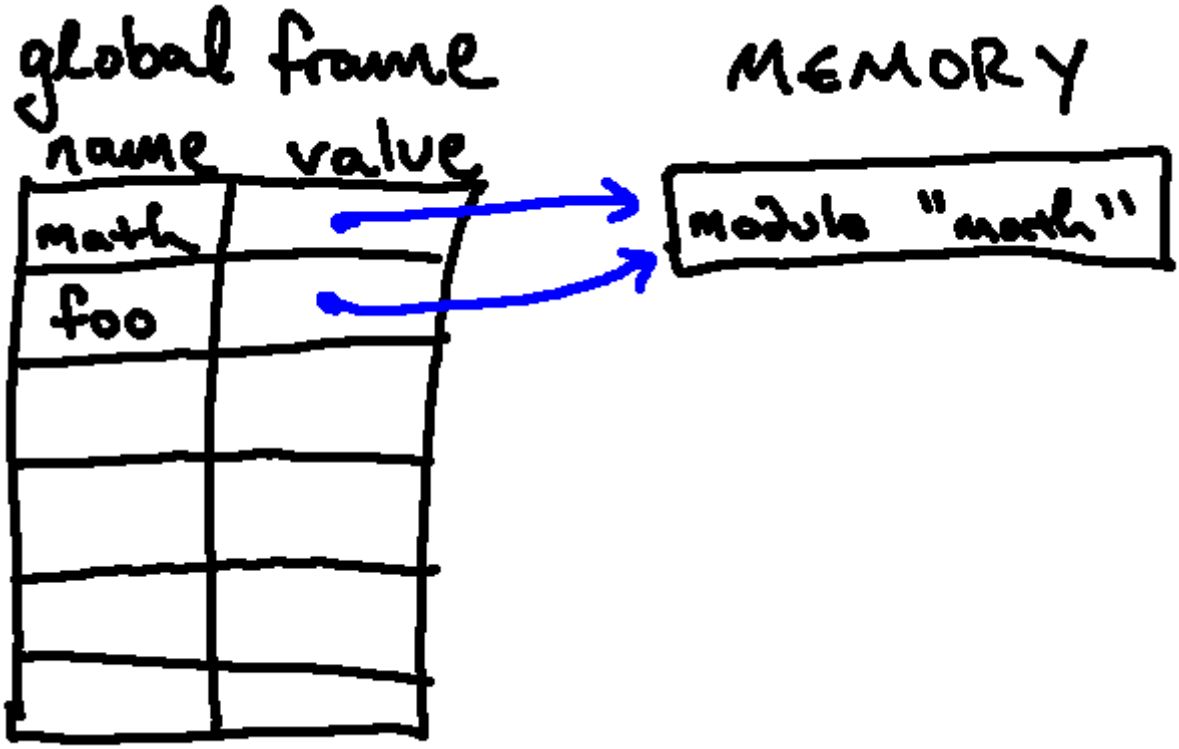
Encontre uma função na documentação do módulo `math` que pode ajudá-lo a atingir esse objetivo (Dica: isso pode ser feito em uma única chamada de função). Use o conteúdo relevante do módulo `math` para escrever um programa curto que calcula e imprime o valor de $\log_3 82$.

▶ Mostrar/Esconder

Objetos `module` , de muitas maneiras, podem ser tratados como quaisquer outros objetos Python: por exemplo, mesmo que talvez não seja uma coisa sensata a se fazer, poderíamos dar outro nome ao módulo `math` após importá-lo:

```
import math
foo = math
```

Isso resultaria no seguinte diagrama de ambiente:



e então poderíamos, por exemplo, procurar `foo.sin` e obter a mesma função `sin` que obteríamos avaliando `math.sin`!

Há muito poder no módulo `math`, e acontece que o Python tem uma série de módulos integrados úteis dos

3) Definição de Funções Personalizadas

Usar funções integradas ou funções importadas de módulos Python é muito bom, mas o *poder real* vem de ser capaz de definir suas próprias funções. Isso é realizado por meio de um novo tipo de instrução Python chamada *instrução de definição de função*, que usa uma nova palavra-chave especial Python chamada `def`.

Isso talvez seja melhor visto por exemplo:

```
def maximum(x, y):
    if x > y:
        z = x
    else:
        z = y
    return z
```

Esta declaração faz duas coisas:

- ele cria um novo objeto `function` na memória, *e*
- ele associa o nome `maximum` a esse objeto no quadro atual.

Uma instrução de definição de função sempre começa com a palavra-chave `def`, seguida por um nome arbitrário para a função. A sequência de nomes entre parênteses após o nome da função são chamados de *parâmetros* (neste caso, `x` e `y`), e a função descreve um cálculo em termos desses parâmetros (bem como, potencialmente, outros valores, constantes, etc.).

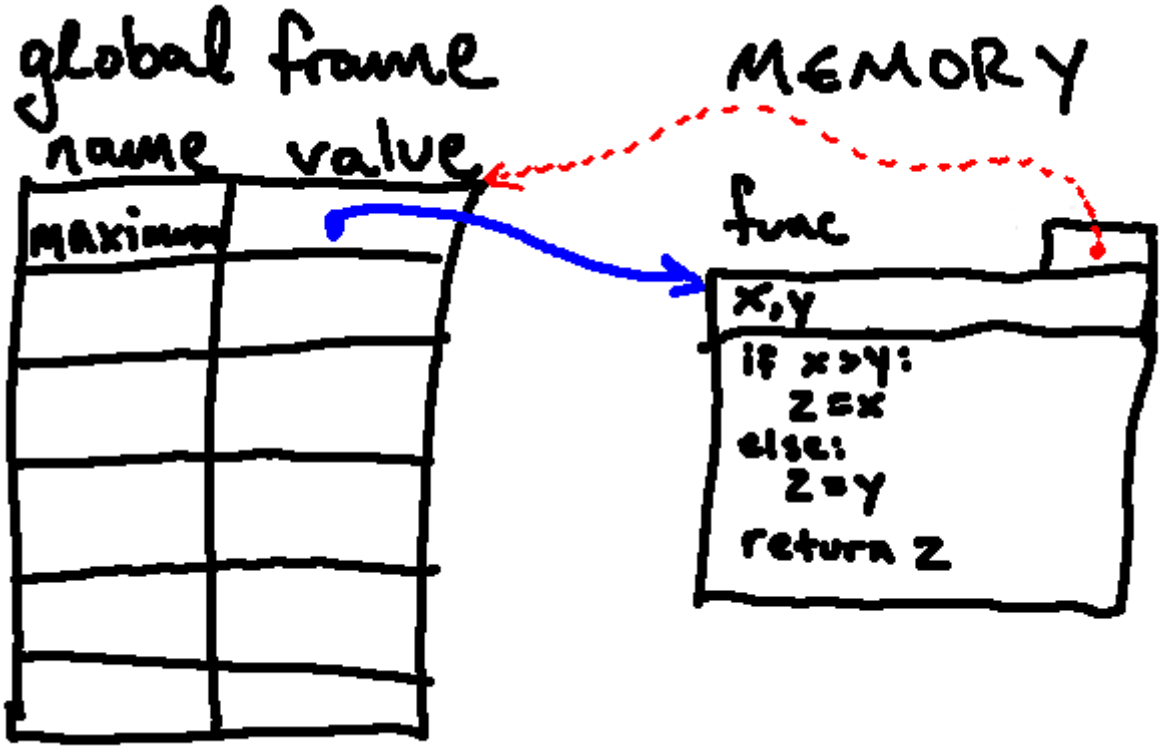
Como muitas das estruturas que vimos, as definições de função também têm um corpo (todo o código que é indentado um nível além de `def`; neste caso, toda a instrução if/else). A palavra-chave `return`, que só pode ser usada dentro de uma definição de função, diz ao Python o que a função deve produzir como seu *valor de retorno* quando for chamada.

É importante ressaltar que essa instrução apenas *define* a função; Python **ainda não executa o código no corpo da função!** Ele simplesmente cria um objeto que representa essa função e associa o nome fornecido a esse objeto.

Como acontece com todos os novos objetos que introduzimos, precisaremos encontrar uma maneira de representar esses objetos na memória. As funções precisam rastrear três informações, e tentaremos descrever todas elas em nossa representação:

1. Os nomes dos parâmetros da função, em ordem;
2. O código no corpo da função; e
3. O quadro em que a função foi definida.

O seguinte mostra um exemplo de diagrama de ambiente que resultaria após a execução da instrução de definição de função acima:



Algumas notas sobre este desenho:

- Observe que a definição da função fez duas coisas: criou um novo objeto de função e ligou o nome `maximum` a esse objeto no quadro global.
- Os nomes `x`, `y` na linha superior do objeto de função representam os parâmetros da função.
- A seta vermelha aponta de volta para o quadro no qual a função foi originalmente definida (neste caso, o quadro global).

Podemos então chamar esta função como faríamos com qualquer uma das funções integradas (ou importadas) que vimos até agora. Por exemplo:

```
a = 7.0
b = 8.0
x = 3.0
y = 4.0

c = maximum(a, b)
```

Agora vamos passar algum tempo pensando sobre o que acontece quando essa função é chamada. Em termos mais simples, Python faz o seguinte:

1. Python procura o nome `maximum` e encontra o objeto de função na memória.
2. Python executará o código no corpo da função com os parâmetros substituídos pelos argumentos fornecidos (aqui, o corpo da função seria avaliado com `x` substituído por `7.0` e `y` substituído por `8.0`) até uma declaração `return` ou o final de o corpo é alcançado. Se uma instrução `return` foi alcançada, a execução na função para e o valor associado é retornado; se o fim da função for alcançado (sem atingir uma instrução `return`), a função retorna `None`.

Portanto, após executar o código acima, `c` terá o valor `8.0`.

Dito isso, será importante entendermos *como exatamente o Python chegou a esse resultado* e, portanto, entraremos em detalhes na próxima seção. Então pegue um copo de chá ou café e acomode-se! A próxima

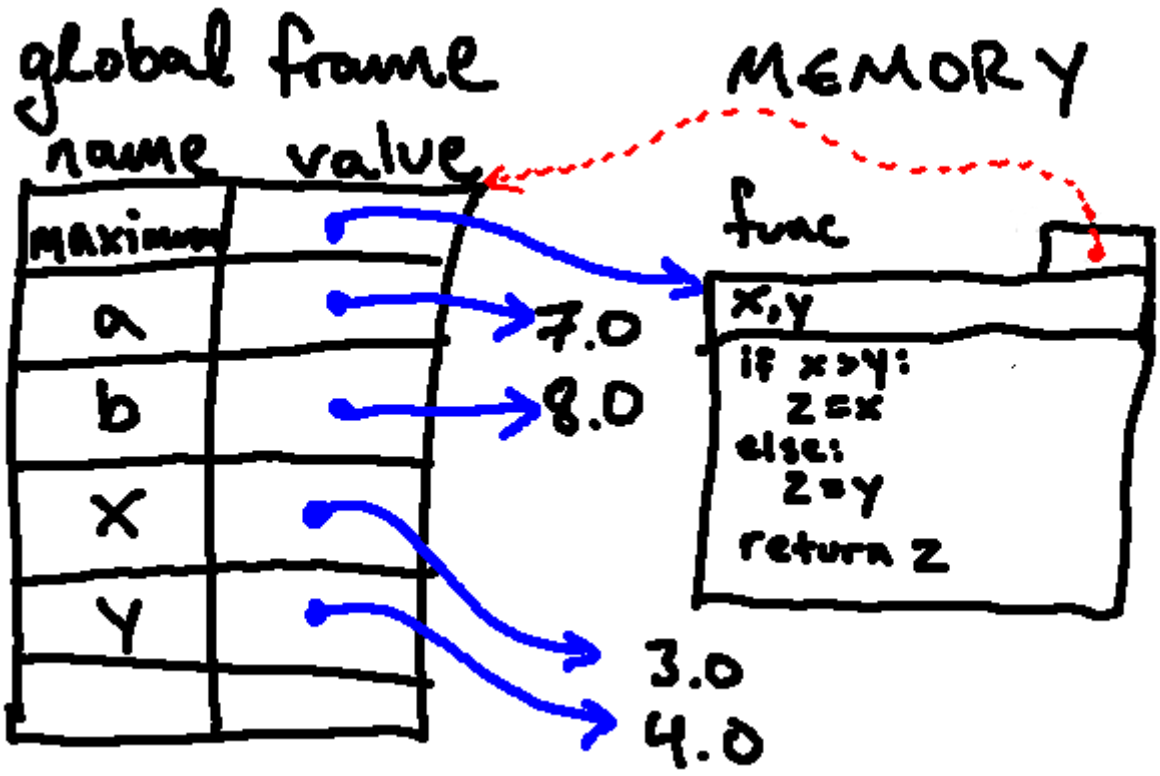
4) Invocação de Funções Personalizadas

Nesta seção, examinaremos o que acontece quando uma função definida pelo usuário é chamada. Para começar, vamos explicar o processo que ocorre, usando o exemplo acima como um exemplo de execução. Depois disso, veremos alguns outros exemplos para ter uma noção dos efeitos desse processo.

Aqui está o código (repetido acima) que usaremos como nosso exemplo de execução (suponha que `maximum` já tenha sido definido como acima):

```
a = 7.0
b = 8.0
x = 3.0
y = 4.0

c = maximum(a, b)
```

Sabemos como as primeiras quatro linhas se comportarão: Python associará os nomes `a`, `b`, `x` e `y` aos valores `7.0`, `8.0`, `3.0` e `4.0`, respectivamente, na memória, resultando no seguinte diagrama de ambiente:



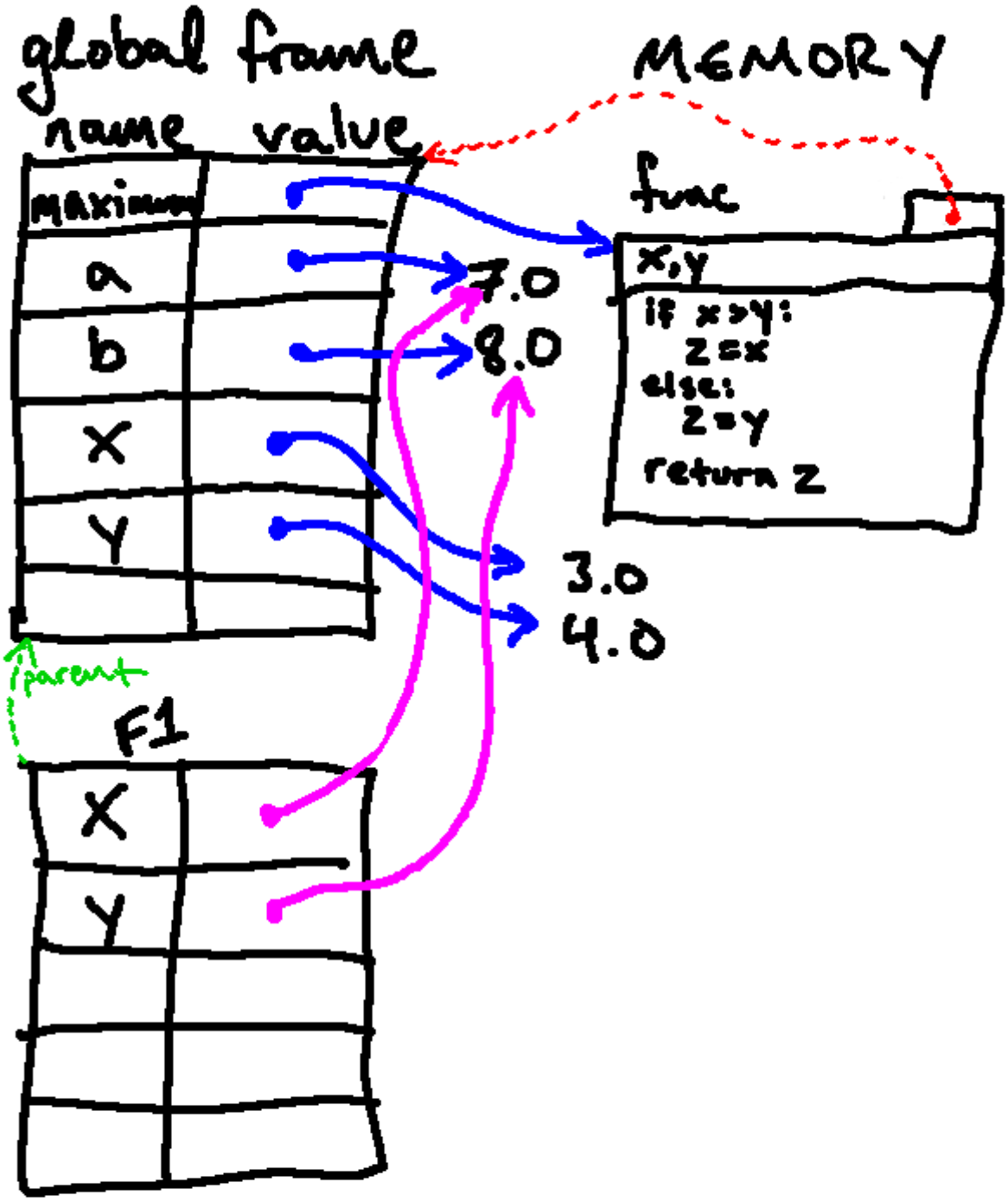
Agora estamos na linha onde a chamada de função é avaliada. Para realizar essa avaliação, o Python conclui as seguintes etapas:

1. Como vimos com as funções integradas, o Python primeiro começa avaliando o nome `maximum` (encontrando o objeto `function`), seguido pelos argumentos para a função (que avaliam `7.0` e `8.0` para os quais `a` e `b`, respectivamente, apontam).
2. É aqui que as coisas ficam diferentes. A próxima etapa do Python ao chamar uma função definida pelo usuário é *criar um novo quadro*. Este quadro será semelhante ao quadro global no sentido de que mapeará nomes para variáveis, mas essas variáveis serão *locais* para a função (elas só estarão acessíveis dentro da função que está sendo chamada). Abaixo, vamos rotular esse novo quadro como **F1**.

Depois que esse novo quadro é criado, o Python associa os nomes dos parâmetros ao argumento que foi passado para a função. Deste ponto em diante, as pesquisas de variáveis acontecerão dentro deste novo quadro (até que a execução da função seja concluída).

Este quadro também contém um "ponteiro parental" ("parent pointer") para *o ambiente no qual a função sendo chamada foi definida*. Nesse caso, já que `maximum` foi definido no quadro global, o ponteiro parental desse novo quadro será para o quadro global. Mas como é possível que funções sejam definidas dentro de funções, nem sempre será o quadro global. Especificamente, será sempre o quadro em que a função foi definida.

Depois que essa etapa for concluída, teremos um diagrama de ambiente como o seguinte:

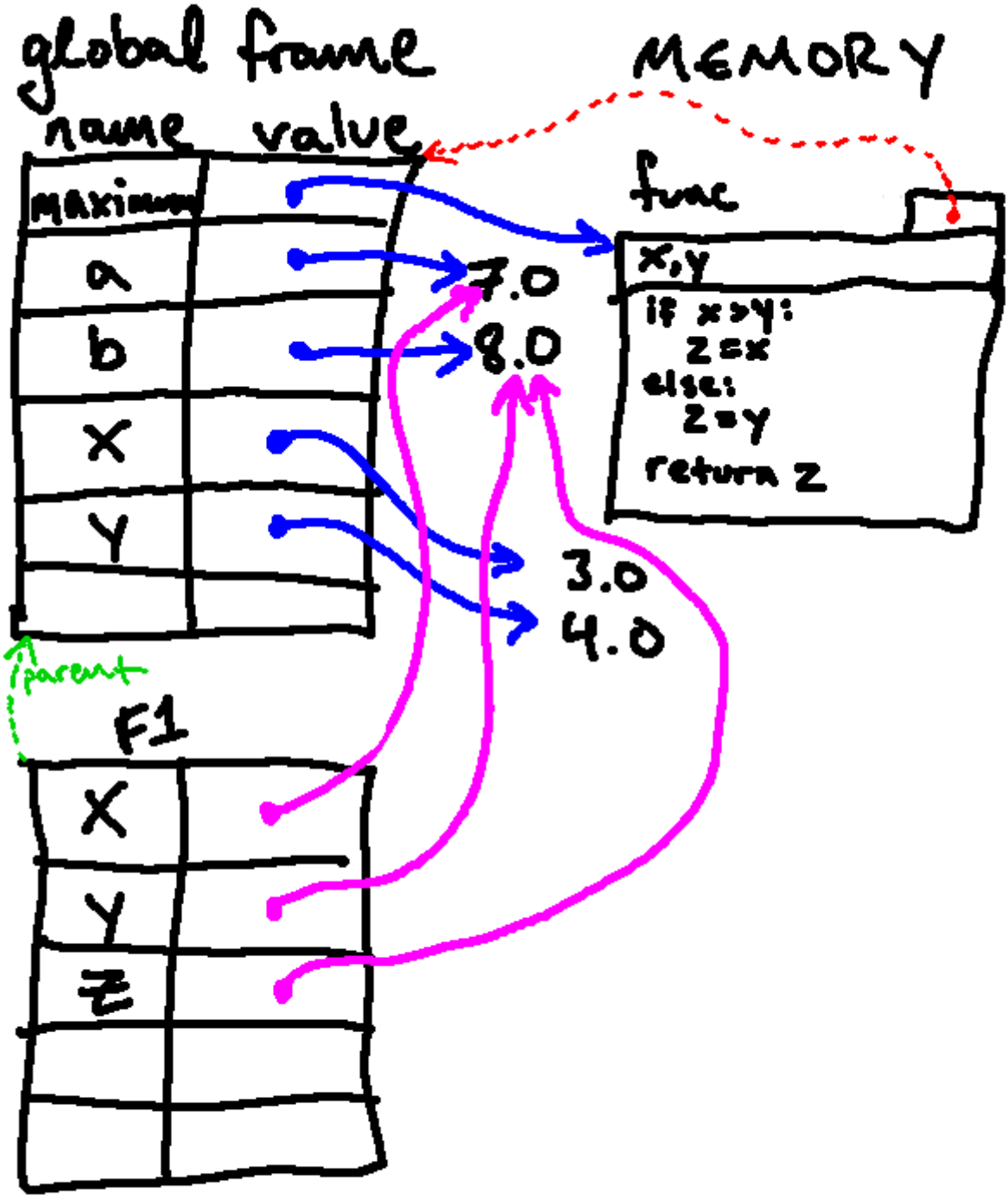


Como prometido, isso é um pouco complicado. O mapeamento na parte inferior esquerda representa as ligações que existem dentro da função. A seta verde representa o "ponteiro parental".

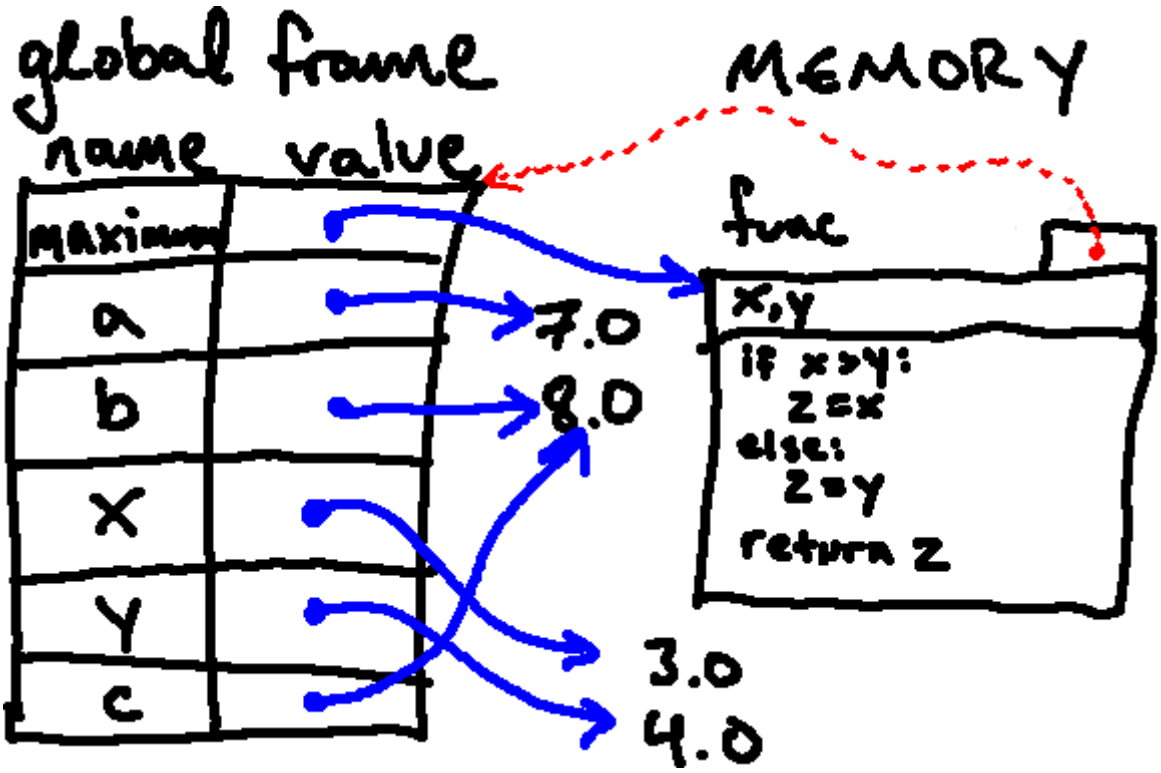
1. Python então executa o corpo da função dentro desse novo quadro. Isso significa que, se Python procurar `x` ao ser executado em **F1**, ele encontrará o valor `7.0` (vinculado em **F1**), em vez do valor `3.0` vinculado no quadro global. Ao pesquisar uma variável, se ela não for encontrada no quadro atual, o Python continua procurando por esse nome no ambiente parental antes de desistir.

   No processo de execução do corpo da função, se o Python fizer novas atribuições, essas *também* serão feitas no quadro atual (nesse caso **F1**).

   Portanto, quando nosso código de exemplo atribui um valor à variável `z`, essa ligação é feita no quadro atual (**F1**). No decorrer da execução da condicional, atribuímos `z` ao mesmo valor de `y`, o que resulta no seguinte diagrama de ambiente:



1. Quando o corpo termina ou uma instrução `return` é alcançada, Python anota o valor a ser retornado (observe aqui que o "return" vermelho não indica uma nova variável real chamada "return"; em vez disso, simplesmente indica o valor que deve ser ser retornado da função). Aqui, o valor de retorno foi `z`, que apontou para o `8.0` atualmente na memória, então esse é o valor que estará disponível como resultado da chamada desta função:



Em seguida, Python para de executar esta função e volta a executar no quadro a partir do qual a função foi chamada, após retornar o valor em questão. Ele também limpa o novo quadro criado. Em nosso exemplo acima, o valor de retorno é então associado ao nome `c` no quadro global (porque estávamos executando `c = maximum(a, b)`), deixando-nos com o seguinte:

4.1) Abstração

Observe que o processo de criação deste novo ambiente nos deu um recurso maravilhoso: as variáveis *dentro* do corpo da função não afetam as variáveis *fora* do corpo da função. Isso nos permitiu chamar algo `x` dentro do corpo da função sem causar problemas com a coisa chamada `x` *fora* da função (quando a função termina de ser executada, se imprimirmos `x`, vemos o `3.0` que foi atribuído a `x` no quadro global)!

Este é um exemplo de como as funções reais são tão poderosas: nos oferecem um meio de *abstração* (ou seja, uma vez que definimos uma função, podemos usá-la sabendo apenas seu comportamento *end-to-end*, sem nos preocuparmos exatamente *como* isso comportamento foi implementado no corpo da função, quais nomes de variáveis foram usados, etc.

4.2) Resumo

Aqui está uma versão resumida de como ilustrar o processo de chamar uma função definida pelo usuário (novamente, para obter detalhes, consulte acima):

1. Avalie os operandos. Se um operando é uma chamada de função, aplique este procedimento a ele
2. Desenhe um novo quadro. Rotule-o com o seguinte:
   - algum indicador único na parte superior do quadro (usaremos nomes como **F** seguido por um número inteiro crescente)
   - um ponteiro para o quadro pai (o quadro no qual a função foi definida)
3. Vincule os parâmetros. Na etapa 1, você já avaliou e simplificou os operandos. Agora você só precisa vincular variáveis a esses valores no novo quadro.
4. Execute o corpo da função neste novo quadro. Dependendo do que você vê, você desenhará mais ligações ou novos quadros.

5. Observe o valor de retorno da função.
6. Quando a execução terminar, remova o quadro e retome a execução no quadro de chamada.

É muito importante conhecer bem essas etapas!

5) Funções são Objetos

Falaremos mais sobre isso no próximo conjunto de leituras, mas uma coisa importante a observar é que em Python, funções são objetos como qualquer outro objeto. Considere o seguinte pequeno trecho de código:

```
def add_one(x):
    return x + 1

do_something = add_one
```

Depois de executar este código, teremos dois nomes no quadro global (especificamente, `add_one` e `do_something`), e ambos os nomes serão associados a um único objeto de função. Portanto, depois de executar esse código, `add_one` e `do_something` referem-se ao mesmo objeto de função na memória.

Como tal, poderíamos invocar a função, por exemplo, com `add_one(3)` *ou* `do_something(3)`, e Python continuaria exatamente da mesma maneira em ambos os casos.

6) Mais Exemplos

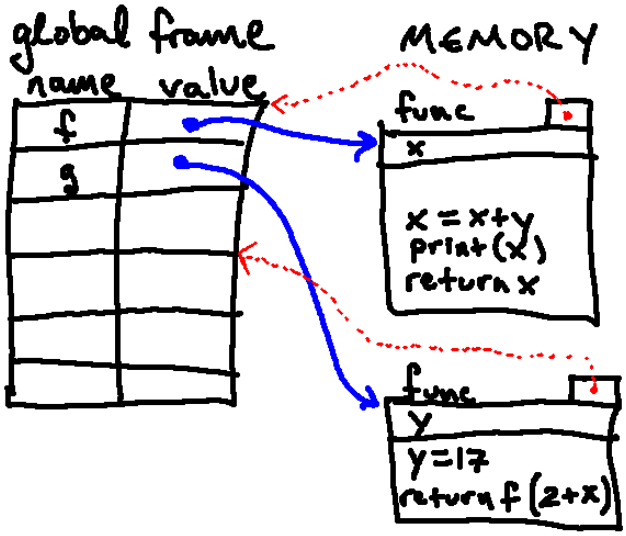
Nesta seção, exploraremos mais dois exemplos, que mostram como esse processo funciona em exemplos um pouco mais complicados.

6.1) Chamando Função dentro de outra Função

Como um segundo exemplo, vamos percorrer o seguinte trecho de código (reconhecidamente bobo):

```
def f(x):
    x = x + y
    print(x)
    return x

def g(y):
    y = 17
    return f(2 + x)

x = 3
y = 4
z = f(6)

a = g(y)

print(z)
print(a)
print(x)
print(y)
```

Tente usar um diagrama de ambiente para prever quais valores serão impressos na tela enquanto este programa é executado. Você pode percorrer nossa explicação de como esse código é executado usando os botões abaixo:



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

PASSO 1

As instruções de definição criam duas novas funções e ligam-nas a `f` e `g`, respectivamente, resultando no diagrama acima.

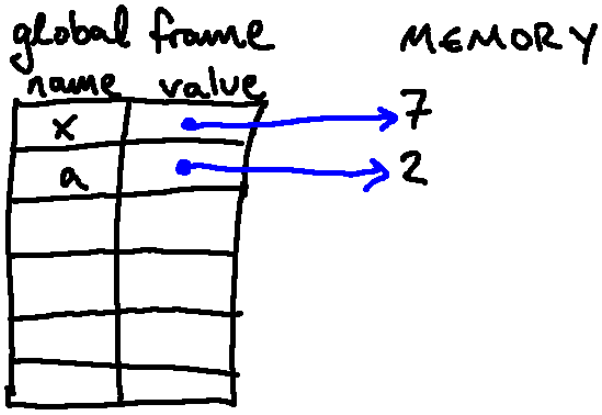
Até aqui, nenhum valor foi exibido ainda.

6.2) Definindo uma Função dentro de uma Função

Como um exemplo final (para este conjunto de leituras, pelo menos!), vamos percorrer o seguinte trecho de código. Este trecho de código demonstra uma nova ideia: como os corpos das funções podem conter código arbitrário, eles também podem incluir *outras definições de função*! Considere o seguinte código:

```
x = 7
a = 2

def foo(x):
    def bar(y):
        return x + y + a
    z = bar(x)
    return z

print(foo(14))
print(foo(27))
```

Tente usar um diagrama de ambiente para prever quais valores serão impressos na tela enquanto este programa é executado. Você pode percorrer nossa explicação de como esse código é executado usando os botões abaixo:



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

PASSO 1

As duas primeiras atribuições ligam `x` a `7` e `a` a `2`, resultando no diagrama acima.

---

7) Built-Ins (Integrados)

Agora que falamos sobre a noção de quadros múltiplos, podemos esclarecer algo sobre os recursos integrados do Python. Falamos muito sobre como o Python procura os valores associados aos nomes de variáveis, e você pode ter se perguntado como o Python encontrou os valores embutidos.

Como Python sabia qual função chamar quando referenciamos `print`? É verdade que `print` e os outros integrados não existem no quadro global. Em vez disso, podemos pensar no próprio quadro global como tendo um ponteiro pai para um quadro especial "built-ins": quando o Python procura um nome no quadro global e não o encontra, ele procura neste "built-ins" frame antes de lançar um `NameError`.

É assim que as pesquisas por, por exemplo, `print` e `len` ocorrem: Python primeiro procura por eles no quadro global. Visto que não os encontra lá, ele procura no quadro de integrados, onde os encontra.

Pesquisas de variáveis que falham também procedem da mesma maneira. Digamos que cometemos um erro de digitação e estamos procurando acidentalmente o valor `pritn`. Ao pesquisar isso, Python iria primeiro olhar no quadro global; quando ele não encontra `pritn` lá, ele então olha no quadro de integrados; e quando não encontrar um `pritn` lá, ele desistirá e levantará um `NameError`.

7.1) Um Lembrete

Se você acha esses diagramas entediantes, não está sozinho! No final das contas, há um motivo pelo qual queremos que os computadores sejam os responsáveis por isso, afinal; eles são muito melhores nessas operações do que nós, e muito mais rápidos! Portanto, no curto prazo, esse tipo de trabalho é entediante. Mas os benefícios de longo prazo são ótimos! Esse tipo de prática é realmente útil na construção de nosso modelo mental do comportamento do Python, o qual é importante ter em mente para que mais tarde, quando encontrar um comportamento inesperado, você possa voltar ao modelo. Com a prática, esse tipo de pensamento se tornará instintivo, e você não terá que desenhar esses diagramas com tantos detalhes (você será capaz de fazer isso em sua cabeça!). Mas, por enquanto, desenhe-os!

8) Resumo

Nesta leitura, apresentamos várias novas ideias. Introduzimos *funções* e *chamadas* de funções integradas do Python. Falamos sobre como importar funções e constantes de módulos Python (como `math`).

Também falamos sobre a *definição de novas funções de sua própria criação* como um meio de abstrair os detalhes de uma computação específica para que ela possa ser reutilizada. Gastamos muito tempo e esforço concentrando-nos em como definir e chamar essas funções se encaixam em nossos modelos mentais de Python (modelo de substituição e diagramas de ambiente).

Nesse conjunto de exercícios, você vai ganhar prática com como simular a avaliação de funções, como importar e usar constantes e funções, e como definir funções próprias!

No próximo conjunto de leituras e exercícios, continuaremos a nos aprofundar no amplo mundo de