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

Na última seção, apresentamos um meio muito poderoso de abstração em Python: *funções*, que nos permitiu abstrair os detalhes de uma computação específica para que ela pudesse ser calculada várias vezes em diferentes entradas com relativa facilidade. Gastamos muito tempo na última seção sobre os detalhes de como o Python interpretava funções e, em particular, a questão de *scoping* (de decidir como e onde o Python procura um nome de variável em particular); nesta seção, dedicaremos mais tempo a maneiras interessantes de usar funções. Também apresentaremos outro novo tipo de objeto Python (um *dicionário*) e falaremos um pouco sobre métodos gerais para projetar programas.

1) Funções são de Primeira Classe ¶

Um recurso poderoso do Python é que ele trata as funções como objetos de primeira classe (http://en.wikipedia.org/wiki/First-class_function), o que significa que as funções no Python podem ser manipuladas de várias maneiras que outros dados podem (especificamente, podem ser passados como argumentos para outras funções, definidas dentro de outras funções, retornado de outras funções e atribuído a variáveis). Nesta seção, exploraremos como podemos usar esse recurso em nossos programas.

1.1) Definindo Funções

A maneira mais comum de definir funções em Python (ou, pelo menos, da maneira que já vimos) é por meio da palavra-chave `def`. Por exemplo, se quisermos fazer uma função que duplique sua entrada, poderíamos fazer isso da seguinte maneira:

```
def double (x):
    return 2 * x
```

Isso criará um novo objeto de função e também associará o nome `double` a esse objeto. Observe que isso não *executa* a função; se quisermos fazer isso, podemos *chamar* a função, por exemplo:

```
print(double(7))  # exibe 14
```

Quando Python executa este código, ele:

1. Procura `double` no ambiente atual
2. Avalia as expressões entre parênteses (neste caso, `7`)
3. *Chama* `double` com o único argumento `7`. Nesse caso, como o nome `double` foi mapeado para uma função, isso foi bem-sucedido.

Python tem outra maneira de definir funções: a palavra-chave `lambda`. Outra maneira de fazer uma função que duplica sua entrada é com:

```
lambda x: 2*x
```

Isso cria uma função quase exatamente igual a `double`, exceto que não tem um nome. `lambda` nunca é necessário (você pode SEMPRE usar `def` no lugar), mas é conveniente em algumas situações (veremos exemplos em breve).

Observe que podemos operar *nas próprias funções* da mesma forma que faríamos com outros objetos Python. Por exemplo, podemos atribuir outro nome à função `double`:

```
other_name = double
print(other_name(20))  # 40
```

Depois de executar `other_name = double`, tanto `other_name` quanto `double` são associados ao mesmo objeto de função, portanto, avaliar `other_name(20)` e avaliar `double(20)` produzem exatamente o mesmo resultado.

1.2) Funções como Argumentos

Imagine que você queira fazer gráficos de várias funções diferentes. Uma etapa desse processo seria descobrir quais valores "y" correspondem a cada um dos vários valores "x". O código a seguir calcula essas respostas para diferentes funções:

```
import math

def sine_response(lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(math.sin(i))
        i = i + step
    return out

def cosine_response(lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(math.cos(i))
        i = i + step
    return out

def double_response(lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(double(i))
        i = i + step
    return out

def square_response(lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(i**2)
        i = i + step
    return out
```

Agora imagine que você executou essas funções, mas decidiu que queria mudar a maneira como estava fazendo o teste. Do jeito que está agora, isso seria uma dor, porque você teria que alterar manualmente cada uma dessas funções. No entanto, podemos corrigir isso criando uma função geral chamada `response`, que recebe uma função como entrada e retorna a lista de suas saídas no intervalo especificado:

```
def response(func, lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(func(i)) # aqui, aplicamos a função fornecida a i
        i = i + step
    return out
```

Observe que, dentro da definição de `response`, chamamos `func` (qualquer função que foi passada como argumento). Usando esta função, poderíamos calcular a resposta de nossa função `double` anterior:

```
# esses dois calculam a mesma resposta!
out = double_response(0, 1, 0.1)
out = response(doulb, 0, 1, 0.1)
```

Observe que quando passamos `double` como um argumento, não colocamos parênteses depois dele. Isso ocorre porque queremos nos referir à própria função (que é chamada de `double`), e não a qualquer saída particular da função (que obteríamos chamando-a, como em `double(7)`).

Tente agora:

Considere a seguinte definição de função:

```
def square (x):
    return x * x
```

Qual é o tipo de cada um dos seguintes valores?

- `square(4)`
- `square(-0.2)`
- `square()`
- `square`

▶ Mostrar/Esconder

Se não nos importássemos em poder acessar `double` fora do cálculo de sua resposta, poderíamos usar `lambda` em vez de criar `double` com `def` (observe que isso tem o mesmo resultado de passar uma *função* como o primeiro argumento para `response`; esta função só está sendo definida com `lambda` em vez de com `def`):

```
# isso calcula a mesma resposta acima!
# se não quisermos usar a função em outras situações, pode
# faz sentido usar lambda em vez de dar um nome à função.
out = response(lambda x: 2*x, 0, 1, 0.1)
```

Observe que poderíamos calcular respostas para todas as funções descritas acima usando esta nova função `response`:

```
sine_out = response(math.sin, 0, 1, 0.1)
cosine_out = response(math.cos, 0, 1, 0.1)
double_out = response(double, 0, 1, 0.1)
square_out = response(lambda x: x**2, 0, 1, 0.1)
```

1.2.1) Exemplo

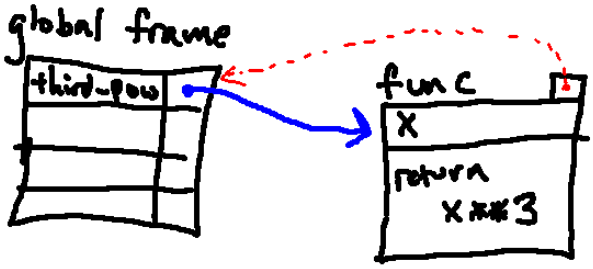
Vamos dar uma olhada na execução de um desses exemplos com mais detalhes para ver como o Python lida com isso. Em particular, vamos percorrer a visualização do diagrama de ambiente do seguinte código:

```
def third_pow(x):
    return x ** 3

def response(func, lo, hi, step):
    out = []
    i = lo
    while i <= hi:
        out.append(func(i))
        i = i + step
    return out

a = response(third_pow, 1, 3, 2)
print(a)
```

Abaixo está uma sequência de diagramas de ambiente explicando o processo pelo qual o Python avalia o código acima:



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

PASSO 1

Python começa avaliando a primeira definição de função, que cria um novo objeto de função e o associa com o nome `third_pow` no quadro global, resultando no diagrama acima.

1.3) Retornando Funções

Outro recurso útil é que funções podem não apenas ser passadas como argumentos para as funções, mas também podem ser retornadas como resultado da chamada de outras funções! Imagine que temos as seguintes funções, cada uma projetada para adicionar um número diferente à sua entrada:

```
def add_1(x):
    return x + 1

def add_2(x):
    return x + 2
```

Se quisermos fazer muitos desses tipos de funções, seria bom ter uma maneira automatizada de fazê-los, em vez de definir cada nova função manualmente. Podemos fazer isso em Python com:

```
def add_n(n):
    def inner(x):
        return x + n
    return inner
```

Isso pode ser um pouco difícil de entender no início, mas o que está acontecendo é o seguinte: quando `add_n` for chamado, ele fará uma *nova função* (aqui, chamada de `inner`) usando a palavra-chave `def`, e então retornará esta função.

Aqui está um exemplo do uso desta função (incluindo usá-la para recriar `add_1` e `add_2` acima):

```
add_1 = add_n(1)
add_2 = add_n(2)

print(add2(3))  # exibe 5
print(add1(7))  # exibe 8
print(add_n(8)(9))  # exibe 17
```

Tente agora:

Qual é o tipo de cada um dos seguintes valores?

- `add_n`
- `add_n(7)`
- `add_n(9)(2)`
- `add_n(0.2)(3)`
- `add_n(0.8)(2)`

▶ Mostrar/Esconder

Observe que também poderíamos ter definido `add_n` da seguinte forma, usando `lambda`:

```
def add_n(n):
    return lambda x: x + n
```

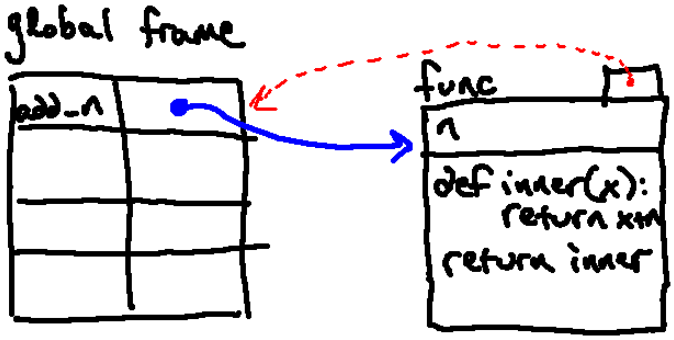
1.3.1) Diagramas de Ambiente

Os exemplos acima podem ser um pouco surpreendentes, mas podemos entendê-los trabalhando usando um diagrama de ambiente. Mesmo que não sejam surpreendentes, é importante observar exatamente o que o Python está fazendo nos bastidores. Aqui, veremos como simular uma parte do código acima usando um diagrama de ambiente:

```
def add_n(n):
    def inner(x)
        return x + n
    return inner

add1 = add_n(1)
add2 = add_n(2)

print(add_2(3))
print(add_1(7))
```



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

PASSO 1

A primeira declaração de definição cria uma função `add_n` , conforme mostrado no diagrama acima. Observe que, como antes, definir a função *não* faz com que o corpo da função seja executado. Como tal, ainda não atingimos a definição de `inner` (na verdade, não vamos acertar até *chamar* `add_n`).

1.4) Resumo

Nesta seção, exploramos a ideia de funções como *objetos de primeira classe*, um recurso que abre algumas portas interessantes em termos de design de programas (veremos alguns exemplos disso neste conjunto de exercícios). A seguir, iremos discutir uma ferramenta realmente poderosa para projetar programas, chamada *recursão*.

2) Recursão

Nota

Recursão é uma ferramenta muito poderosa, mas pode ser um tópico muito confuso/intimidante para programadores iniciantes e leva um tempo para começar a parecer natural. Então, se demorar um pouco para realmente internalizar essa noção, isso é totalmente normal.

Se, depois de ler esta seção, você não entender imediatamente como recursão funciona ou como projetar programas que fazem uso de estruturas recursivas, não se desespere! Com o tempo e com mais exposição (neste curso e além), essas estruturas começarão a parecer naturais; e com mais prática, será mais fácil usar a recursão em seus próprios programas. Continue tentando e, mais importante, venha pedir ajuda se ficar perdido com essas leituras e/ou os exercícios associados!

Como vimos no último conjunto de leituras, é possível que uma função chame outra. Acontece que também é possível que uma função chame a si mesma! Pode não ser imediatamente óbvio por que isso é bom, mas acontece que ter essa capacidade torna a expressão de certos tipos de cálculos muito mais fácil.

Vamos começar com um pequeno exemplo e, no final desta seção, teremos alguns exemplos mais realistas e interessantes.

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

Se `n` for 0 ou negativo, esta função imprime a palavra "Blastoff!" Caso contrário, ele imprime `n` e, em seguida, chama uma função chamada `countdown` (ou seja, ela *se chama*), passando `n-1` como um argumento. O que acontece se chamarmos essa função assim?

```
countdown(3)
```

De uma perspectiva de alto nível, a execução deste programa procede da seguinte forma:

- A execução de nossa chamada original para `countdown` começa em um novo quadro com `n=3` e, como `n` é maior que 0, ele exibe o valor 3 e, em seguida, chama a si mesmo ...
  - No processo de execução de nossa chamada original, chamamos `countdown` *novamente*. Como antes, a execução dessa nova chamada à `countdown` começa em um novo quadro com `n=2` e, como `n` é maior que 0, ela produz o valor `2` e, em seguida, chama a si mesmo ...

        - A execução de mais uma chamada a `countdown` começa em um novo quadro com `n=1` e, como `n` é maior que 0, ela produz o valor `1` e então se chama ...
            - A execução de outra chamada para `countdown` começa em um novo quadro com `n=0` e, como `n` não é maior que 0, ela produz a palavra "Blastoff!" e então retorna (neste caso, porque não havia `return`, ele simplesmente retorna `None`).
        - A chamada de função de `countdown` com `n=1` então retorna.
    - A chamada de função de `countdown` com `n=2` então retorna.
- A chamada de função de `countdown` com `n=3` então retorna.

E então voltamos ao corpo principal do programa. Portanto, a saída total é:

```
3
2
1
Blastoff!
```

Uma função que chama a si mesma é chamada de função *recursiva*, e o processo pelo qual ela executa sua execução é chamado de *recursão*.

2.1) Outro Exemplo

Como outro exemplo de recursão, podemos escrever uma função que imprime uma string `n` vezes:

```
def print_n(s, n):
    if n <= 0:
        return # nota: se não dermos um valor para a instrução return, a
 função retorna None
    print(s)
    print_n(s, n-1)
```

Se `n <= 0`, a *instrução de retorno* sai da função. O fluxo de execução retorna imediatamente ao chamador e as linhas restantes da função não são executadas.

O resto da função é semelhante a `countdown`: ela exibe `s` e, em seguida, chama a si mesma para exibir `s` $n - 1$ vezes adicionais. Portanto, o número de linhas de saída é `1 + (n - 1)`, que soma `n`.

Para pequenos exemplos como este, provavelmente é mais fácil usar um loop `for`. Mas veremos exemplos mais tarde que são difíceis de escrever com um loop `for` e fáceis de escrever com recursão, por isso é bom começar cedo e praticar.

2.1.1) Uma Dualidade

Observe que a função acima produz exatamente a mesma saída de um programa que poderíamos ter escrito com um loop `for`:

```
def print_n(s, n): # solução "recursiva"
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)

def print_n (s, n): # solução "iterativa" (usando loops)
    for i in range(n):
        print(s)
```

Você pode ficar tentado a se perguntar: qual é melhor, iteração ou recursão?

Você poderia argumentar que, nesse caso, a solução iterativa é melhor: é mais curta e mais fácil de ler. Mas mais tarde, veremos exemplos onde o oposto é verdadeiro! Para o bem ou para o mal, não há resposta certa se iteração ou recursão é melhor *no sentido geral*; depende tanto do programa que está sendo escrito quanto das preferências pessoais do programador.

Algumas pessoas preferem escrever em um estilo *iterativo* (usando loops), alguns preferem escrever em um estilo *recursivo* (usando recursão), e quase todos irão escrever em ambos os estilos, uma vez que alguns cálculos podem ser mais fáceis de pensar ou expressar em um determinado estilo (independentemente de preferência pessoal).

Em geral, é possível reescrever qualquer programa iterativo usando recursão e *vice-versa*. Mas às vezes é muito mais fácil expressar um programa em um estilo do que no outro. Com o tempo, você desenvolverá uma noção do que funciona melhor para você em quais situações.

2.2) Exemplo: Fibonacci

Alguns cálculos matemáticos são (relativamente) facilmente especificados por meio de *indução*, que está intimamente relacionada à recursão: elas têm algum conjunto de casos básicos e o resto dos resultados são baseados nesses casos básicos.

Considere, por exemplo, a sequência de Fibonacci: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

Essa sequência é naturalmente expressa recursivamente (ou seja, é naturalmente expressa em termos de si mesma). Se $F[n]$ é o n-ésimo número de Fibonacci e n é um número inteiro não negativo, então uma representação concisa da sequência é dada por:

$$F[n] = \begin{cases} n & \text{if } \ n < 2 \\ F[n-1] + F[n-2] & \text{otherwise} \end{cases}$$

A primeira forma é o que chamamos de *caso base*. Um caso base é um caso relativamente simples, que pode ser resolvido sem recursão. A segunda forma especifica o *caso geral* (ou o *caso recursivo*), que por sua vez depende da definição de F e que tem a propriedade de reduzir em direção a um caso base.

Esta é uma forma totalmente geral para uma definição recursiva, ou seja, todas as definições recursivas devem ter um ou mais *casos base* e um ou mais *casos recursivos*.

Uma implementação recursiva de um programa para calcular o n-ésimo número de Fibonacci reflete esta forma:

```
def fib (n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

A maneira como o Python procederá com a avaliação é basicamente a mesma que um ser humano faria ao usar a forma matemática acima. Se pensássemos, por exemplo, em calcular $F[3]$ à mão, poderíamos proceder da seguinte forma, sempre expandindo o termo mais à esquerda que pode ser expandido:

$$\begin{aligned} F[3] &= F[2] + F[1] \\ &= F[1] + F[0] + F[1] \\ &= 1 + F[0] + F[1] \\ &= 1 + 0 + F[1] \\ &= 1 + 0 + 1 \\ &= 2 \end{aligned}$$

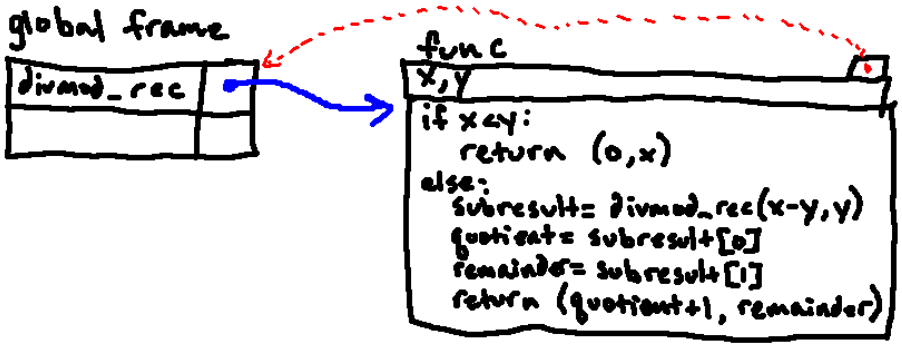
Python avaliará `fib(3)` de maneira semelhante. Para descobrir como exatamente cada função está sendo chamada e seus resultados calculados, poderíamos desenhar um diagrama de ambiente. No entanto, aqui, estamos interessados em um comportamento de nível superior e, por isso, abstrairemos os detalhes internos da função `fib` abaixo, focando em vez disso no comportamento *end-to-end*.

- Quando chamamos `fib(3)`, Python eventualmente chega a uma instrução que o informa para retornar `fib(2) + fib(1)`. Para fazer isso, ele deve avaliar `fib(2)` e `fib(1)`.
  - Avaliando `fib(2)`, Python eventualmente chega a uma instrução que o informa para retornar `fib(1) + fib(0)`. Para fazer isso, ele deve avaliar `fib(1)` e `fib(0)`.
    - Avaliar `fib(1)` retorna `1`, e
    - avaliar `fib(0)` retorna 0.
  - Com esses resultados, Python sabe que `fib(2)` será `1`.
  - Agora, avaliando `fib(1)`, Python vê que retorna `1`.
- Com esses resultados, Python sabe que `fib (3)` retorna sua soma, ou `2`.

2.3) Diagramas de Ambiente

É importante ressaltar que recursão não exige que mudemos nada em nosso modelo de ambiente, e Python não trata as funções recursivas de maneira diferente de outras funções. Em vez disso, a recursão funciona *por causa* das regras que já definimos em leituras anteriores. Aqui, veremos um exemplo de diagrama de ambiente para uma função recursiva. Em particular, vamos dar uma olhada no exemplo a seguir, que é uma implementação recursiva do programa `divmod` que escrevemos na semana passada (que realiza a divisão inteira de `x` e `y` fornecidos e retorna uma tupla contendo o quociente e o restante):

```
def divmod_rec(x, y):
    if x < y:
        return (0, x)
    else:
        subresult = divmod_rec(x-y, y)
```



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

PASSO 1

Primeiro, avaliamos a definição da função, que cria um novo objeto de função e o associa com o nome `divmod_rec` no global frame, resultando no diagrama mostrado acima.

A seguir, avaliaremos `divmod (179, 67)`.

2.4) Projetando Funções Recursivas

Em geral, desenvolver uma função recursiva envolve a escolha de casos base sensatos e a determinação de como uma solução geral para outros casos pode ser representada em termos da própria função. Nesta seção, vamos percorrer o processo envolvido no projeto de duas funções recursivas. Como mencionamos acima, recursão é um tópico especialmente complicado (por vários motivos, incluindo alguns que expandiremos a seguir). Portanto, se você não conseguir encontrar as respostas para as perguntas abaixo por conta própria e se for necessário ler algumas vezes esses exemplos e/ou pedir a ajuda de alguém, tudo bem!

2.4.1) Fatorial

Primeiro, consideraremos escrever uma função recursiva para calcular o *fatorial* de um determinado inteiro não negativo. Então, vamos pensar em escrever uma função `factorial` que seja definida recursivamente.

Tente agora:

Vamos começar pensando em um caso base apropriado (ou seja, uma entrada para a qual não precisamos de recursão para calcular a resposta). Tente pensar em algumas entradas para as quais podemos retornar uma resposta imediatamente.

▶ Mostrar/Esconder

Tente agora:

Agora vamos considerar o caso recursivo. Se recebemos um número que não é um de nossos casos básicos, como podemos usar uma chamada recursiva para nos ajudar a calcular a saída adequada?

▶ Mostrar/Esconder

Tente agora:

Tente traduzir isso em um programa Python funcional. Teste seu código em alguns casos. Você também pode achar útil que sua função imprima seu argumento como uma primeira etapa, para ver como as chamadas recursivas estão funcionando.

▶ Mostrar/Esconder

2.4.2) Nivelamento de Lista

Como um segundo exemplo, consideraremos o problema de "nivelar" uma lista: dada uma lista (que pode conter outras listas), retorne uma nova lista contendo todos os valores do original, em ordem, mas remove qualquer aninhamento de listas.

Por exemplo:

```
flatten_list([1, 2, [3, [4, [[[5]]]], 6, [7, 8]], 9])
```

retornaria:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Acontece que existem várias maneiras de pensar sobre este programa. Vamos explorar uma dessas abordagens em profundidade aqui, mas é importante notar que existem outras.

Tente agora:

Vamos começar pensando em um caso base apropriado. Como mencionamos acima, um caso base é aquele em que não precisamos de recursão para encontrar a resposta.

Tente pensar em alguns exemplos de entrada para os quais não precisaríamos de recursão para calcular uma lista simples. Você pode generalizar isso para um caso básico?

▶ Mostrar/Esconder

Tente agora:

Agora, vamos tentar pensar no caso recursivo. Dada uma lista que não se enquadra no caso base acima, como poderíamos calcular a saída adequada, possivelmente fazendo uso de uma ou mais chamadas recursivas para `flatten_list`?

▶ Mostrar/Esconder

A solução esboçada acima faz uso da verificação de tipo, que é algo que não discutimos nessas leituras até agora, portanto, falaremos sobre isso brevemente aqui. Faremos uso de uma função integrada chamada `type` ao implementar `flatten_list`. `type` recebe um objeto arbitrário como entrada e retorna seu tipo. Por exemplo, `type(2)` retorna `int`. Portanto, podemos verificar se um determinado valor `x` é um inteiro da seguinte maneira: `type (x) == int`. Faremos uso desta função a seguir.

Tente agora:

Isso é um pouco desafiador, mas tente traduzir as ideias acima em um programa Python e tente executá-lo em algumas entradas para ver se funciona.

▶ Mostrar/Esconder

2.5) Casos Base e Recursão Infinita

Observe que cada uma das funções recursivas que examinamos até agora teve pelo menos um caso base (em que nenhuma chamada recursiva é feita). Se uma recursão nunca atinge um caso base, ela continua fazendo chamadas recursivas para sempre e o programa nunca termina. Isso é conhecido como *recursão infinita* e geralmente não é uma boa ideia. Aqui está um programa mínimo com uma recursão infinita:

```
def recurse():
    recurse()
```

Na maioria dos ambientes de programação, um programa com recursão infinita não funciona para sempre. Ao executar este programa, de fato vemos um erro:

```
    File "broken_recursion.py", line 2, in recurse
    File "broken_recursion.py", line 2, in recurse
    File "broken_recursion.py", line 2, in recurse
                  .
                  .
                  .
    File "broken_recursion.py", line 2, in recurse
RuntimeError: profundidade máxima de recursão excedida
```

Por padrão, Python se limita a chamadas recursivas que vão até 1000 "camadas" de profundidade. Portanto, quando o erro ocorre, existem 1000 quadros `recurse` ! Se você encontrar uma recursão infinita por acidente, revise sua função para confirmar se há um caso base que não faz uma chamada recursiva. E se houver um caso base, verifique se é garantido que você irá alcançá-lo.

2.6) Resumo

Nesta seção, exploramos a noção de *recursão*, um conceito complicado (mas poderoso) pelo qual definimos funções em termos de si mesmas.

Em geral, uma solução recursiva consiste em um ou mais casos base (que produzem uma saída diretamente), bem como um ou mais casos recursivos (que produzem uma saída com base na saída de uma ou mais chamadas para a própria função).

Vimos alguns exemplos de funções recursivas acima e também falamos um pouco sobre o processo de projetar funções recursivas.

3) Dicionários

Agora vamos mudar um pouco de direção para aprender sobre outro tipo integrado chamado *dicionário*. Dicionários são um dos melhores recursos do Python; eles são os blocos de construção de muitos programas eficientes e elegantes.

De certa forma, um dicionário é como uma lista, mas é mais geral. Listas e dicionários são objetos compostos e mutáveis. Em uma lista, porém, os índices devem ser inteiros, enquanto em um dicionário eles podem ser (quase) qualquer tipo.

3.1) Um Dicionário é um Mapeamento

Um dicionário contém uma coleção de índices, que são chamados de *chaves* (*keys*), e uma coleção de valores. Cada chave está associada a um único *valor* (*value*). A associação de uma chave e um valor é chamada de par de *chave-valor* (*key-value*) ou, às vezes, de *item*.

Em linguagem matemática, podemos dizer que um dicionário representa um *mapeamento* de chaves para valores, então você também pode dizer que cada chave "mapeia" para um valor. Como exemplo, construiremos um dicionário que mapeia palavras do inglês para o alemão, de forma que as chaves e os valores sejam todos strings.

Listas em Python são criadas com colchetes. Dicionários, por outro lado, são criados com chaves (`{}`). Por exemplo, poderíamos fazer um dicionário vazio com:

```
en2de = {}
```

Para adicionar itens ao dicionário, você pode usar colchetes:

```
en2de['cat'] = 'Katze'
```

Esta linha cria um item que mapeia da chave `'cat'` para o valor `'Katze'` . Se imprimirmos o dicionário agora, veremos um par de valores-chave com dois pontos entre a chave e o valor:

```
print(en2de)  # imprime {'cat': 'Katze'}
```

Este formato de saída também é um formato de entrada. Por exemplo, você pode criar um novo dicionário com três itens:

```
en2de = {'cat': 'Katze', 'guinea pig': 'Meerschweinchen', 'dog': 'Hund'}
```

Observe que os elementos de um dicionário não são pesquisados com base em sua ordem; em vez disso, você usa as chaves para pesquisar os valores correspondentes:

```
print(en2de['dog'])  # imprime 'Hund'
```

A chave `'dog'` sempre mapeia para o valor `'Hund'` , então a ordem dos itens não importa muito!

Se a chave não estiver no dicionário, você receberá um erro:

```
print (en2de ['eagle']) # nos dá um erro: KeyError: 'eagle'
```

Até agora, vimos que colchetes são usados para pesquisar valores em um dicionário e também para adicionar novos itens a um dicionário. Eles também são usados para *alterar* um mapeamento existente. Por exemplo, se o idioma alemão mudasse repentinamente de modo que "cat" se traduzisse a "Loewchen", poderíamos atualizar nosso dicionário para levar em conta essa mudança:

```
en2de['cat'] = 'Loewchen'
print (en2de)  # imprime {'cat': 'Loewchen', 'dog': 'Hund', 'guinea pig':
'Meerschweinchen'}
```

Mas isso não é provável que aconteça, então vamos colocá-lo de volta:

```
en2de['cat'] = 'Katze'
```

3.2) Chaves e Valores Válidos

É importante ressaltar que os dicionários não se limitam a conter strings. Eles podem ter objetos arbitrários como valores e objetos *imutáveis* arbitrários como chaves. Também podemos misturar, portanto, um único dicionário pode ter chaves e/ou valores de vários tipos diferentes.

Portanto, qualquer um dos tipos que aprendemos até agora são *valores* válidos (incluindo listas, funções, outros dicionários, etc), mas certos objetos não podem ser *chaves* para um dicionário (de forma simplificada, objetos mutáveis como listas e outros dicionários não podem ser chaves em um dicionário).

Tente agora:

Experimente associar uma chave *mutável* a um novo valor no dicionário `en2de`, por exemplo:

```
x = [14,7,5]
en2de[x] = 2
```

O que acontece?

▶ Mostrar/Esconder

O que acontece se você tentar usar um dicionário como chave? Uma tupla?

▶ Mostrar/Esconder

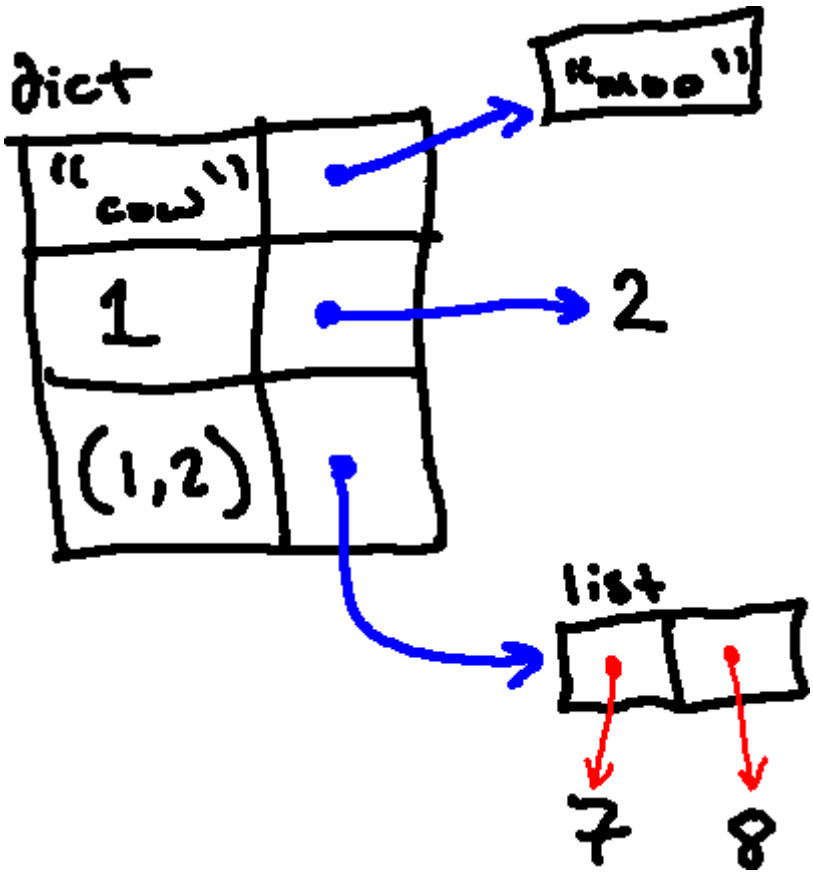
3.3) Dicionários em Diagramas de Ambiente

Como sempre fizemos quando introduzimos um novo tipo de objeto Python, precisaremos de uma maneira de representar dicionários em diagramas de ambiente. Observe que um dicionário é muito parecido com um quadro em alguns aspectos; é um mapeamento entre chaves e valores. Portanto, nossa representação será muito semelhante: colocaremos as chaves no lado esquerdo deste mapeamento e o lado direito consistirá em ponteiros para os valores associados.

Por exemplo, o seguinte dicionário:

```
{'cow': "moo", 1: 2, (1, 2): [7, 8]}
```

seria representado da seguinte forma em um diagrama de ambiente:



3.4) Outras Operações: Comprimento

Além das operações acima (adicionar, alterar e pesquisar pares de chave-valor), também podemos fazer uma série de outras operações interessantes em dicionários.

A função `len` funciona em dicionários; ele retorna o número de pares de chave-valor:

```
print(len(en2de))  # imprime 3
```

3.5) Outras Operações: o Operador "in"

Também temos outra operação útil em dicionários (e, de fato, em muitos dos outros objetos compostos que estudamos) por meio do operador `in` .

Com os dicionários, o operador `in` informa se algo aparece como uma *chave* no dicionário (aparecer como um valor não é bom o suficiente).

```
print('cat' in en2de)  # imprime True
print('sandwich' in en2de)  # imprime False
print('Katze' in en2de)  # imprime False
```

3.5.1) "in" para Outras Sequências

O operador `in` também funciona para outros tipos de objetos compostos, conforme descrito abaixo:

- `element in some_list` é avaliado como `True` se *qualquer objeto equivalente a* `element` (comparado por `==`) existe como um dos elementos na lista representada por `some_list` .
- `element in some_tuple` se comporta da mesma maneira.
- `some_string in some_other_string` se comporta de maneira diferente: será avaliada como `True` se a string `some_string` for uma *substring* da string `some_other_string` (uma subsequência contígua)

Tente agora:

Tente prever o resultado da avaliação das seguintes expressões em Python para ter uma ideia de como `in` se comporta nesses tipos de objetos. Use Python para verificar você mesmo.

- `7 in [4,9,7]`
- `7.0 in [4,9,7]`
- `7 in "67"`
- `'cat' in {'species': 'cat', 'name': 'Fluffy'}`
- `'species' em {'species': 'cat', 'name': 'Fluffy'}`
- `'sn' em 'parsnips'`
- `'tummy' em 'tomato'`

3.5.2) O operador "not in"

Python também fornece um operador chamado `not in` , que faz a verificação oposta, mas é mais fácil de escrever (e ler) do que a forma alternativa. Portanto, as duas expressões a seguir são equivalentes:

```
not needle in haystack # usando o operador "in" e usando "not" no resulta
do
needle not in haystack # usando o operador "not in"
```

3.6) Exemplo: Dicionário como uma Coleção de Contadores

Suponha que você receba uma string e queira contar quantas vezes cada letra aparece. Existem várias maneiras de fazer isso:

1. Você pode criar 26 variáveis, uma para cada letra do alfabeto. Em seguida, você pode percorrer a string e, para cada caractere, incrementar o contador correspondente, provavelmente usando uma condicional encadeada.
2. Você pode criar uma lista com 26 elementos. Em seguida, você pode converter cada caractere em um número, usar o número como um índice na lista e incrementar o contador apropriado.
3. Você pode criar um dicionário com caracteres como chaves e contadores como valores correspondentes. Na primeira vez que você ver um caractere, você adicionaria um item ao dicionário. Depois disso, você aumentaria o valor de um item existente.

Cada uma dessas opções executa o mesmo cálculo, mas cada uma delas implementa esse cálculo de uma maneira diferente.

Por exemplo, uma vantagem da implementação do dicionário é que não precisamos saber com antecedência quais letras aparecem na string e apenas temos que abrir espaço para as letras que aparecem.

Aqui está uma possibilidade de código:

```
def histogram(string):
    d = {}
    for char in string:
        if char not in d:
            d[char] = 1
        else:
            d[char] = d[char] + 1
    return d
```

O nome da função é `histogram`, que é um termo estatístico para uma coleção de contadores (ou frequências).

A primeira linha da função cria um dicionário vazio. O loop `for` itera sobre a string. Cada vez através do loop, se o caractere `char` não estiver no dicionário, criamos um novo item com a chave `char` e o valor inicial `1` (já que vimos esta letra uma vez). Se `char` já estiver no dicionário, incrementamos `d[char]`.

Por exemplo:

```
h = histogram('brontosaurus')
print(h)  # imprime {'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

O histograma indica que as letras `'a'` e `'b'` aparecem uma vez; `'o'` aparece duas vezes e assim por diante.

3.7) Outras Operações: "get"

O padrão acima de verificar se uma chave existe em um dicionário e usar um valor padrão (`0` no exemplo acima) se a chave não existir é um padrão muito comum e, portanto, Python oferece uma maneira mais fácil de atingir esse mesmo objetivo, com `get`.

`get` recebe uma chave e um valor padrão como entradas. Se a chave aparecer no dicionário, `get` retorna o valor correspondente; caso contrário, ele retorna o valor padrão. Por exemplo:

```
h = histogram('a')
print(h)  # {'a': 1}
print(h.get('a', 0))  # 1
print(h.get('b', 0))  # 0
```

Tente agora:

Use `get` para escrever `histogram` de forma mais concisa. Você deve ser capaz de eliminar a instrução `if`.

▶ Mostrar/Esconder

3.8) Looping e Dicionários

Se você usar um dicionário em uma instrução `for`, ele percorre as chaves do dicionário. Por exemplo, a função `print_hist` abaixo imprime cada chave e o valor correspondente:

```
def print_hist(hist):
    for char in hist:
        print(char, hist[char])
```

Podemos executá-lo da seguinte maneira:

```
h = histogram('parrot')
print_hist(h)
```

e produzirá a seguinte saída:

```
a 1
p 1
r 2
t 1
o 1
```

Novamente, as chaves não estão em uma ordem específica. Para percorrer as chaves em ordem, você pode usar a função integrada `sorted`:

```
for key in sorted(h):
    print(key, h[key])
```

produz a seguinte saída:

```
a 1
o 1
p 1
r 2
t 1
```

3.9) Exemplo: Pesquisa Reversa

Dado um dicionário `d` e uma chave `k`, é fácil encontrar o valor correspondente `v = d[k]`. Essa operação é chamada de *lookup*.

Mas e se você tiver `v` e quiser encontrar a(s) chave(s) associada(s) `k`? Você tem dois problemas:

- Primeiro, pode haver mais de uma chave mapeada para o valor `v`, ou pode não haver nenhuma!
- Em segundo lugar, não existe uma sintaxe simples para fazer um *lookup inverso*. Você precisa procurar

4) Projetando Programas

Até agora, passamos muito tempo apresentando os blocos de construção que o Python nos oferece e relativamente pouco tempo falando ativamente sobre como projetar um bom programa. Fizemos isso pelo que acreditamos ser um bom motivo (na verdade, como podemos esperar fazer o Python fazer consistentemente o que queremos se não temos um entendimento sólido de como ele responde a várias entradas e do que é capaz?), mas agora que construímos um subconjunto suficientemente avançado do Python, vale a pena olhar na outra direção também: como alguém pode ir de uma declaração de problema para um programa funcional?

Nesta seção, falaremos um pouco sobre o que constitui bom estilo em um programa e, em seguida, mudaremos o foco para falar sobre uma estratégia para projetar programas. A boa notícia é que você já tem alguma prática em ambas as áreas e, por isso, tentaremos aproveitar essa experiência aqui.

4.1) Estilo

Quando falamos sobre *estilo* de programação neste texto, normalmente não estamos nos referindo a escrever código que *parece* bonito, mas sim a escrever código que seja tão fácil de ler, escrever, entender e debug quanto possível.

O que constitui um bom estilo e bom design é, de certa forma, uma questão subjetiva. No entanto, existem algumas noções relacionadas ao estilo que são razoavelmente bem aceitas:

- **Nomes Importam**: escolher bons nomes para funções, parâmetros e outras variáveis ajudará a tornar seu código muito mais fácil de entender. Os nomes das funções devem descrever o que fazem. Nomes de variáveis devem descrever o que representam (não apenas seus tipos!). Nomes de uma única letra são aceitáveis em algumas situações, mas use esses nomes com moderação.
- **Don't Repeat Yourself (DRY)**: (também conhecido como *menos é mais*) Vários fragmentos de código não devem descrever lógica redundante. Em vez disso, essa lógica deve ser simplificada em um loop, função ou variável (dependendo do tipo específico de repetição com que você está lidando). Se você se pegar reescrevendo a mesma expressão curta repetidamente, isso pode ser um bom sinal de que você pode armazenar isso em uma variável como um resultado intermediário. Se você estiver copiando/colando um bloco de código para calcular um resultado, essa pode ser uma oportunidade para definir uma função.
- **Generalidade Ganha**: é melhor definir funções e programas de maneira geral e permitir que suas entradas tratem de casos específicos. Por exemplo, a função `square` não é definida no módulo `math`, em parte porque é um caso de uso específico para a função `pow` (exponetiação), que está no módulo `math`.
- **Planeje para mudança**: de muitas maneiras, esta é uma extensão de *Generalidade Ganha*. Frequentemente, ao escrever programas, você pode descobrir que os requisitos do programa que está escrevendo podem mudar (ou pode descobrir que o problema que realmente queria resolver não é aquele resolvido pelo programa que está escrevendo). Como tal, sempre que possível, é importante tentar fazer programas que sejam (relativamente) fáceis de mudar, caso seja necessário.

Essas diretrizes ajudarão a:

- melhorar a legibilidade do seu código
- reduzir o número de erros em seu código
- tornar mais fácil fazer alterações em seu programa se você precisar
- minimizar a quantidade de código que você escreve Na verdade, este último objetivo é importante, embora "a sabedoria convencional reverencie a complexidade". Você pode ouvir falar de pessoas falando sobre seus programas em termos de "linhas de código" como métrica, mas na direção errada.

Um bom objetivo geralmente é minimizar a quantidade de código escrito para resolver um problema específico (as pessoas costumam se surpreender com o quão pequeno um programa poderoso pode ser!). Bill Gates resumiu essa ideia muito bem:

```
"Medir o progresso de programação por linhas de código é como medir o pro
gresso da construção de aeronaves por peso."
- Bill Gates
```

4.1.1) Exemplo: Um "Filtro de Média"

Neste exemplo, falaremos sobre como refinar sucessivamente um programa para melhorá-lo em termos de estilo. Começaremos com um trecho de código que possui vários problemas e o melhoraremos gradualmente ao longo desta seção.

Imagine escrever um programa para aplicar um "filtro de média" a uma lista de números. Podemos pensar sobre isso como aplicando uma média móvel: isto é, se tivermos uma lista de entrada com n números $x_0, x_1, \ldots, x_n$, queremos calcular novos valores $y_k, y_{k+1}, \ldots, y_{n-(k-1)}$ de modo que cada valor na lista de saída seja a média dos k valores anteriores na lista de entrada:

$$y_i = \frac{x_i + x_{i-1} + \cdots + x_{i-(k-1)}}{k}$$

Vamos começar calculando a média de corrida para a lista a seguir, com $k = 3$:

```
input_list = [100, 27, 93, 94, 107, 10]
```

Como $k = 3$, não podemos calcular os valores apropriados para as duas primeiras entradas, portanto, nossa lista de saída deve ter 4 elementos neste caso.

Aqui está uma maneira de começar a escrever um programa para calcular o resultado desejado:

```
## Programa número 1
input_list = [100, 27, 93, 94, 107, 10]

averaged_result = [0] * 4

averaged_result[0] = (input_list[0] + input_list[1] + input_list[2]) / 3
averaged_result[1] = (input_list[1] + input_list[2] + input_list[3]) / 3
averaged_result[2] = (input_list[2] + input_list[3] + input_list[3]) / 3
averaged_result[1] = (input_list[3] + input_list[4] + input_list[5]) / 3

print(averaged_result)
```

Esse é o tipo de código que pode surgir ao escrever uma linha de código e, em seguida, copiá-lo/colá-lo, fazendo pequenas alterações para dar conta das diferenças no que queríamos calcular em cada etapa.

Os mais minuciosos devem ter notado que dois bugs conseguiram se infiltrar nas versões coladas: o último resultado não está sendo armazenado no local adequado e o penúltimo resultado não está sendo calculado corretamente!

Essa classe de erros (esquecer de fazer uma alteração ou fazer uma alteração incorreta em código copiado/colado) é uma ocorrência relativamente comum, chamada (logicamente) de "Erro de Copiar/Colar". Muitas vezes podemos evitar esses tipos de erros reestruturando para evitar repetição.

Este é um caso em que podemos perceber, com relativamente pouco esforço, que estamos nos repetindo muito. Aqui, estamos realizando uma operação várias vezes para diferentes elementos em uma lista e, portanto, podemos "refatorar" este código para fazer uso de um loop:

```
## Programa número 2
input_list = [100, 27, 93, 94, 107, 10]

averaged_result = []
for i in range(2, 6):
    this_avg = (input_list[i - 2] + input_list[i - 1] + input_list[i]) /
3
    averaged_result.append(this_avg)

print(averaged_result)
```

De muitas maneiras, esse é um código muito mais interessante. Mas agora vamos imaginar que quiséssemos realizar esse mesmo cálculo em uma *segunda* lista de números. Uma opção disponível para uso é copiar/colar esta estrutura e fazer alterações com base nas propriedades da nova lista:

```
## Programa número 3
input_list = [100, 27, 93, 94, 107, 10]

averaged_result = []
for i in range(2, 6):
    this_avg = (input_list[i - 2] + input_list[i - 1] + input_list[i]) /
3
    averaged_result.append(this_avg)

print(averaged_result)

input_list2 = [94, 38, 96, 20, 18, 100, 108]

averaged_result2 = []
for i in range(2, 7):
    this_avg = (input_list[i - 2] + input_list[i - 1] + input_list[i]) /
3
    averaged_result.append(this_avg)

print(averaged_result2)
```

Aqui estamos nos repetindo mais uma vez! Estamos realizando exatamente o mesmo cálculo em duas entradas diferentes. Aqui, parece que devemos ser capazes de generalizar essa operação usando uma função! E, de fato, podemos, embora precisemos pensar cuidadosamente sobre alguns detalhes dessa função. O resultado final é mostrado aqui:

```
## Programa número 4
def averaging_filter(inputs):
    result = []
    for i in range(2, len(inputs)):
        avg_val = (inputs[i - 2] + inputs[i - 1] + inputs[i]) / 3
        result.append(avg_val)
    return result

input_list = [100, 27, 93, 94, 107, 10]
print(averaging_filter(input_list))

input_list2 = [94, 38, 96, 20, 18, 100, 108]
print(averaging_filter(input_list2))
```

Por que este código deve ser considerado melhor?

1. Alguém poderia argumentar que o fluxo geral do programa é mais fácil de ler e entender agora, porque talvez seja mais óbvio qual operação estamos realizando em cada uma das duas listas.
2. Se quisermos executar esse cálculo em mais listas, é muito mais fácil fazê-lo agora (principalmente quando nossa função também leva em conta os comprimentos variáveis das listas).
3. Se descobrimos um bug em nossa implementação do procedimento de média, só temos que implementar a correção em um lugar, ao invés de em vários lugares (assim como com erros de copiar/colar, é fácil esquecer de fazer a alteração em um dos lugares!).

Este código já percorreu um longo caminho, mas alguém poderia argumentar que outra etapa vale a pena. Vimos anteriormente que o comportamento do filtro de média pode mudar com base no valor de k (o número de amostras que calculamos). Como tal, favorecendo a generalidade (no caso de decidirmos amanhã que queremos $k = 5$ em vez de $k = 3$, ou que queremos usar diferentes valores de k em cada uma das listas), poderíamos fazer uma alteração na função para tomar outro parâmetro:

```
## Programa número 5
def averaging_filter(inputs, k):
    result = []
    for i in range(k-1, len(inputs)):
        total = 0.0
        for index in range(i-k+1, i+1):
            total = total + inputs[index]
        result.append(total/k)
    return result

input_list = [100, 27, 93, 94, 107, 10]
print(averaging_filter (input_list, 3))

input_list2 = [94, 38, 96, 20, 18, 100, 108]
print(averaging_filter (input_list2, 3))
```

Aqui, aumentamos a generalidade de um trecho de código, mas essa função se tornou mais difícil de ler. Aqui é onde um comentário bem colocado pode ajudar, descrevendo o que a função foi projetada para fazer.

Por meio desse exemplo, tentamos ilustrar o aprimoramento iterativo de um programa para melhorá-lo não em termos de quão correto é, mas em termos de estilo, de acordo com os princípios expostos acima. Ao longo desta seção, mudamos de um programa que é muito difícil de ler, entender, modificar, estender e debug; a um programa que é muito mais fácil na maioria desses aspectos.

4.2) Projeto e Planejamento

Frequentemente, programadores de experiência são capazes de imaginar soluções elegantes para problemas, o que os programadores iniciantes podem ter dificuldade em fazer. No entanto, é importante lembrar que isso tem muito pouco a ver com talento; é tudo uma questão de experiência e é uma habilidade que pode ser aprendida (embora exija muito tempo e prática!).

Uma boa maneira de começar a desenvolver essa habilidade é realmente tentar seguir as diretrizes descritas acima. Quando você está começando, pode ser difícil imaginar essas coisas com antecedência, mas pode ser mais fácil perceber oportunidades de melhorias *depois de implementar algo*. Depois de ter uma solução implementada, procure oportunidades para melhorar o estilo do seu código, conforme discutido acima; no processo de fazer isso, você pode notar alguns padrões úteis para que da próxima vez que estiver trabalhando em um problema semelhante (ou um problema que exige uma estrutura semelhante), você possa pular a etapa intermediária e ir direto para a versão melhorada!

Dito isso, a ideia não é digitação aleatória e, em seguida, tentando melhorar a partir daí. É sempre importante ter um plano primeiro! Aqui, discutiremos uma maneira fundamentada em princípios de formular um plano. Mas, como sabemos, é impossível fazer planos perfeitos o tempo todo, então tentaremos trabalhar em nossa estrutura a ideia que estamos planejando *supondo que as coisas podem não seguir exatamente esse plano*.

Também é verdade que existem certas diretrizes que podem ser seguidas para ajudar a facilitar o processo de design. Algumas pessoas gostam de formalizar esse processo mais do que outras, mas um jeito é dividir as coisas aproximadamente nas seguintes etapas:

1. Entenda o problema
    - Que problema você está tentando resolver?
    - Qual é a entrada e qual é a saída? Como podemos representá-los em Python?
    - Quais são alguns exemplos de relações de entrada/saída? Encontre alguns exemplos específicos que você pode usar para testar mais tarde.
2. Faça um plano
    - Procure a conexão entre a entrada e a saída. Quais são as operações que precisam ser realizadas para produzir a saída? Como você pode construir a saída usando essas operações? Como você pode testar as operações?
    - Que informações, além das entradas, você precisará manter? Que tipos de objetos Python são úteis para esse fim?
    - Você já leu ou escreveu um programa relacionado antes? Nesse caso, partes dessa solução podem ser úteis aqui.
    - Você pode dividir o problema em casos mais simples? Se você não puder resolver imediatamente o problema proposto, tente primeiro resolver uma subparte do problema ou um problema relacionado, mas mais simples (às vezes, um caso mais geral ou mais específico).
    - O seu plano utiliza todos as entradas? Ele produz todas as saídas adequadas?
3. Implementar o Plano
    - Traduza seu plano para Python.
    - Você pode ser capaz de implementar várias das operações importantes como funções individuais.
    - Conforme você avança, considere as diretrizes de estilo acima. Se você repetir um cálculo, pode querer reorganizá-lo agora (em vez de no final).
    - Conforme você avança, verifique cada etapa. Você pode ver claramente que a etapa está correta? Você pode provar que está correto? Como você pode usar esse resultado como parte de um programa maior?
4. Olhe para trás
    - Teste que o programa é correto e seu estilo.

- Para cada um dos casos de teste que você construiu anteriormente, execute-o e certifique-se de ver o resultado esperado. Existem outros casos de teste que você deve considerar?
- Você poderia ter resolvido o problema de uma maneira diferente? Em caso afirmativo, quais são as vantagens e desvantagens da solução que você escolheu?
- Você pode usar o resultado para algum outro problema? Você pode usar estruturas de programação semelhantes para algum outro problema?
- Procure oportunidades para melhorar o estilo do seu código de acordo com as regras discutidas na seção anterior.
  - Os nomes de suas funções e variáveis são concisos e descritivos?
  - Você está repetindo um cálculo em qualquer lugar?
  - Existem funções ou outras partes do seu código que podem ser generalizadas? Uma grande parte deste esboço é sobre quebrar um problema grande em partes menores e mais fáceis de manejar.

É importante notar que isso ainda não torna necessariamente a formulação de um plano fácil, mas, com o tempo, o processo se tornará instintivo. Por enquanto pode ser muito útil trabalhar com o esboço acima em detalhes sempre que você projetar e implementar um programa.

4.2.1) Exemplo: Projetando o Programa da Cifra de César

Nesta seção, examinaremos o problema da Cifra de César do último conjunto de exercícios do ponto de vista das diretrizes descritas acima. Não vamos percorrer todo o processo aqui; em vez disso, vamos nos concentrar em algumas das peças mais interessantes. É importante ressaltar que não esperamos que as respostas a essas perguntas venham facilmente quando você começar; na verdade, podem levar muito tempo. Mas o que é importante é que você esteja fazendo as perguntas certas e formulando um plano antes de mergulhar no código.

1. *Entenda o problema*: aqui, implementaremos a cifra de deslocamento descrita no último conjunto de exercícios. Recebe como entrada uma string contendo algum texto e produz uma string do mesmo comprimento contendo texto criptografado.

   Letras, números e pontuação são tratados de maneira diferente, portanto, queremos ter certeza de que nosso plano funciona em strings que contêm qualquer combinação desses caracteres. Um bom caso de teste é a string `"376 tuna fish?!"`, porque contém todos esses tipos de caracteres. Se o valor de deslocamento for `2`, devemos receber `"598 vwpc hkuj?!"` como uma saída. Também queremos alguns casos de teste que lidam corretamente com os diferentes comportamentos de wrapping dos diferentes tipos de caracteres, então também devemos testar com um deslocamento maior que 26, com um deslocamento negativo e com um deslocamento menor que -26.
2. *Faça um plano*: uma das grandes dificuldades aqui é a diferença entre as maneiras como os diferentes tipos de caracteres são tratados. Portanto, seria bom separar um do outro. Para este fim, podemos implementar uma função para trocar letras especificamente, bem como uma função para trocar números especificamente (não precisamos de uma para pontuação, uma vez que a pontuação não muda entre as duas strings).

   Para diferenciá-los, também pode ser bom ter uma maneira fácil de *detectar* se um caractere é uma letra ou um número, portanto, podemos querer funções adicionais para esse fim.

   Para construir a saída, será importante manter a saída deslocada "até agora". Podemos começar com uma string vazia e adicionar caracteres deslocados à medida que os computamos.

Com isso em mente, podemos esboçar um plano em termos dessas operações:

```
defina o resultado como uma string vazia.
enquanto ainda tenhamos mais caracteres de entrada a considerar, faça o s
eguinte:
    pegue o próximo caractere da entrada
    se o caractere for uma letra:
        mude o caractere de acordo com as regras de letra
    caso contrário, se o caractere for um número:
        mude o caractere de acordo com as regras numéricas
    case contrário,
        não mude o caractere
    em todos os casos, adicione o caractere deslocado ao final da string
 de resultado
retornar a string resultante
```

1. *Implemente o plano*: vamos deixar isso como um exercício para o leitor :) Mas é uma boa ideia pensar em implementar uma função separada para cada uma das operações discutidas acima: uma função que muda um único letra de acordo com as regras da letra, aquela que muda um único número de acordo com as regras do número, etc. Então você pode construir a cifra de César geral em termos dessas funções (em vez de implementar tudo em uma grande função)
2. *Olhe para trás*: neste ponto, passaríamos por nossos casos de teste acima, nos certificando de ver os resultados corretos para cada um. Podemos, neste ponto, notar outros tipos de casos de teste que valeria a pena executar, e executá-los também. Também podemos notar problemas estilísticos (código repetido em particular), que podemos voltar e melhorar.

4.2.2) Teste e Debugging

Uma parte importante desse processo envolve a criação de casos de teste, e é importante escolher bons casos. No entanto, nem sempre é fácil saber o que constitui um bom caso de teste. No entanto, existem duas diretrizes que chegam muito perto de serem universais:

1. Sempre que possível, seus casos de teste devem abranger todas as *ramificações* possíveis em seu código.
    - Se você tiver uma condicional, certifique-se de ter pelo menos um caso de teste que execute cada bloco associado a essa condicional.
    - Se você tiver um loop `while`, certifique-se de ter um caso de teste que ignore o loop completamente, um que passe pelo loop uma vez e outro que passe pelo loop várias vezes.
    - Se você ainda não escreveu o código, ainda pode pensar em coisas semelhantes: quais tipos de entrada provavelmente serão tratados de forma diferente? Certifique-se de incluir esses tipos de entradas em seus testes.
2. Sempre que possível, cada um de seus testes deve tentar testar *uma coisa específica*.
    - Se um de seus testes falhar, ele não deve apenas informar que *algo* está errado, mas também deve ajudá-lo a descobrir *que parte* de seu programa está errada.
    - Para este fim, é útil testar não apenas seu programa geral, mas todas as funções que você definiu ao longo do caminho.

5) Resumo

Neste conjunto de leituras, exploramos algumas das coisas interessantes que podem ser feitas com funções. Em particular, nos concentramos na ideia de que as funções em Python são *objetos de primeira classe*; isso significa que eles podem ser tratados como quaisquer outros objetos em Python, o que significa,

entre outras coisas, que as funções podem ser passadas como argumentos para funções e também podem ser retornadas como resultado de outras funções!

Também introduzimos a ideia de *recursão* e vimos alguns problemas para os quais a solução recursiva talvez seja mais direta do que a solução iterativa.

Também introduzimos dicionários (que são o último objeto integrado do Python que discutiremos) e vimos