Funções I

MC102-2018s1-Aula16-180424

Arthur J. Catto, Ph.D.
ajcatto@g.unicamp.br

24 de abril de 2018

1 Funções

Uma ***função*** é uma estrutura capaz de agrupar e dar nome a uma sequência de comandos que são executados quando ela é chamada.

Funções nos permitem implementar pelo menos dois conceitos fundamentais para a resolução de problemas complexos:

- ***Modularização***, isto é, a decomposição de uma solução complexa em partes mais simples e, portanto, mais fáceis de serem concebidas, implementadas, depuradas e mantidas.
- ***Abstração***, isto é, a possibilidade de tratarmos uma sequência lógica de comandos como uma "caixa-preta" à qual a gente se refere pelo nome. Assim, podemos nos concentrar *no que ela faz* sem nos preocuparmos com *como ela faz*.

Em Python, uma função tem a seguinte estrutura básica:

```
def nome_da_função(lista_de_parâmetros):
    comando_1
    comando_2
    ...
    comando_n
    return resultado_da_função
```

Quando chamada, uma função inicializa sua *lista_de_parâmetros*, executa os comandos que a compõem e termina retornando seu resultado ao ponto de onde foi chamada.

Como veremos nos exemplos a seguir, a *lista_de_parâmetros*, o *resultado_da_função* e o próprio comando ***return*** são elementos opcionais.

1.1 Como definir e chamar uma nova função

Já exercitamos bastante a chamada de funções disponíveis no sistema e a passagem de argumentos para elas. Vamos ver como fazer isso com funções de nossa autoria.

1.1.1 Exemplo: Uma função para calcular o dobro de um número

Uma função para calcular o dobro de um número x poderia ser definida como...

```
In [2]: def dobro(x):
            dois_x = x + x
            return dois_x
```

e poderia ser chamada assim...

```
In [3]: d = dobro(3) + dobro(0.5)
        print(d)
```

```
7.0
```

Na definição de *dobro*, a variável *dois_x* é desnecessária. Poderíamos ter escrito simplesmente...

```
In [4]: def dobro(x):
            return x + x
```

```
In [5]: d = dobro(3) + dobro(0.5)
        print(d)
```

```
7.0
```

CUIDADO

Note que, ao contrário de muitas outras linguagens de alto nível, Python não controla os tipos dos parâmetros e resultados de uma função.

Essa característica de Python faz com que, embora nossa função se chame *dobro* e tenha sido desenhada para *dobrar um valor numérico*, ela não vai reclamar e vai processar "corretamente" um argumento de qualquer tipo onde o operador **+** esteja definido.

```
In [6]: s = dobro('abc')
        print(s)
```

```
abcabc
```

```
In [7]: l = dobro([1, 'a', 2.3])
        print(l)
```

```
[1, 'a', 2.3, 1, 'a', 2.3]
```

1.2 Uma função pode não ter uma *lista de parâmetros*

1.2.1 Exemplo: Ler e validar uma entrada numérica

Queremos ler repetidamente a entrada até receber um inteiro não negativo e retornar esse número.
Uma função com essa finalidade poderia ser definida como...

```
In [8]: def ler_int_não_neg():
            x = None
            while x is None:
                s = input('Digite um inteiro não-negativo: ')
                if s.isnumeric():
                    x = int(s)
            return x
```

```
In [9]: print(ler_int_não_neg())

12345
```

Note que, mesmo quando uma função não tem uma *lista de parâmetros*, o par de parênteses é necessário tanto na definição quanto na chamada.

Curiosidade: Essa função só aceita inteiros não-negativos. Seria possível aceitar um número qualquer? Nesse caso, o método *isnumeric* não seria adequado porque ele rejeitaria um possível sinal à frente do número, bem como um possível ponto decimal.
Uma saída seria usar o par *try... except*, que estudaremos em detalhe mais tarde. O comando *try* permite *experimentar a execução de um ou mais comandos* e, se ocorrer uma exceção, *capturá-la* e dar um tratamento específico a ela. Por exemplo, ...

```
In [10]: %reset -f

         def ler_num():
             while True:
                 s = input('Digite um número qualquer: ')
                 try:
                     x = float(s)
                     break
                 except:
                     pass
             return x

In [11]: print(ler_num())

-12.345
```

Curiosidade: Essa função sempre retorna um *float*. Seria possível retornar um tipo mais preciso? É possível resolver esse problema *aninhando* pares *try... except* e tentar reconhecer a entrada indo do tipo mais restrito para o mais abrangente. Neste caso, primeiro vamos tentar reconhecer a entrada como um *int* e depois, se não funcionar, como um *float*.

```
In [12]: %reset -f

         def ler_num():
             while True:
                 s = input('Digite um número qualquer: ')
                 try:
                     x = int(s)
                     break
                 except:
                     try:
                         x = float(s)
                         break
                     except:
                         pass
             return x
```

```
In [13]: x = ler_num()
         print(type(x), x)

<class 'int'> -12345


In [14]: y = ler_num()
         print(type(y), y)

<class 'float'> -3.1416
```

1.3 Uma função pode produzir um efeito sem retornar qualquer resultado

1.3.1 Exemplo: Exibir um cabeçalho

Quando uma função apenas realiza uma tarefa sem retornar um resultado específico pode-se dispensar o comando **return**. Nesse caso, a função terminará quando se esgotar seu bloco de comandos. Por exemplo, ...

```
In [15]: %reset -f

         def exibir_cabeçalho():
             título = 'A Revolução dos Bichos'.center(30)
             autor = 'George Orwell'.center(30)
             # espaços subbstituídos por pontos só para "enxergar" o resultado
             print(título.replace(' ', '.'))
             print(autor.replace(' ', '.'))

         exibir_cabeçalho()
         print("\nexibir_cabeçalho() terminou")

...A.Revolução.dos.Bichos...
...George.Orwell...

exibir_cabeçalho() terminou
```

1.4 Uma função pode retornar mais do que um resultado

1.4.1 Exemplo: Contar maiúsculas, minúsculas e outros caracteres numa *string*

Uma função pode retornar mais do que um resultado usando uma tupla para isso...

```
In [16]: %reset -f

         maiúsculas = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
         minúsculas = 'abcdefghijklmnopqrstuvwxyz'

         def mai_min(s):
             nmai, nmin, noutros = 0, 0, 0
             for c in s:
                 if c in maiúsculas:
                     nmai += 1
```

```
            elif c in minúsculas:
                nmin += 1
            else:
                noutros += 1
        return nmai, nmin, noutros

    print(mai_min('The Quick Brown Fox...'))
    print(mai_min('The Quick Brown Fox...'.upper()))
    print(mai_min('The Quick Brown Fox...'.lower()))
```

```
(4, 12, 6)
(16, 0, 6)
(0, 16, 6)
```

1.5 Uma função pode chamar outras funções

1.5.1 Exemplo: Calcular a soma dos quadrados de três parâmetros numéricos

Neste caso podemos dividir nosso problema em dois:

- criar uma função $quadrado(x)$ para calcular o quadrado de um número
- criar uma função $soma\_quadrados(a, b, c)$ para resolver o problema proposto usando a função anterior.

Para que *soma_quadrados* possa usar *quadrado*, a definição de *quadrado* deve preceder a definição de *soma_quadrados*, como mostrado a seguir...

```
In [17]: %reset -f
```

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

    def soma_quadrados(a, b, c):
        return quadrado(a) + quadrado(b) + quadrado(c)

    print(soma_quadrados(3, 4, 5))
```

```
50
```

1.6 Uma função pode chamar a si mesma

1.6.1 Exemplo: Calcular o fatorial de um inteiro não-negativo

Dado n inteiro, calcular $n!$, sabendo-se que $n! = 1 \times 2 \times \ldots \times n$, para $n > 0$ e que $0! = 1$.

Desenvolvimento da solução

Vamos colocar a definição do fatorial na forma de uma especificação de função:
$f(n) = 1 \times 2 \times \ldots \times n$

Uma especificação como essa pode ser facilmente expressa em forma recursiva (isto é, referindo-se a si mesma), se notarmos que:
$f(n-1) = 1 \times 2 \times \ldots \times (n-1)$ e, portanto,
$f(n) = f(n-1) \times n$, para $n > 0$.

Toda definição recursiva (ou recorrente) precisa de uma "âncora", isto é, um ponto fixo capaz de interromper a recursividade (ou a recorrência).

Neste caso, como $f(n)$ depende de $f(n-1)$, $f(n-1)$ depende de $f(n-2)$ e assim por diante, uma âncora razoável será $f(0)$ que sabemos ser igual a 1.

A solução, portanto, pode ser modelada por duas expressões:
$f(n) = f(n-1) \times n$, para $n > 0$
$f(0) = 1$

Isso é o bastante para que implementemos a função desejada.

```
In [18]: %reset -f

         def f(n):
             if n > 0:
                 return f(n - 1) * n
             else:
                 return 1

         print('f(0)  =', f(0))
         print('f(1)  =', f(1))
         print('f(5)  =', f(5))
         print('f(10) =', f(10))
```

```
f(0)  = 1
f(1)  = 1
f(5)  = 120
f(10) = 3628800
```

Note que nossa função não "reclama" ao receber um argumento inválido, que lhe seja passado de forma acidental ou intencional.

```
In [19]: print('f(-5)  =', f(-5))
```

```
f(-5)  = 1
```

Uma forma de criar uma certa proteção é utilizando *asserções*.
Uma asserção é uma expressão lógica que se acredita verdadeira, definida num comando **assert**:
- Se a expressão lógica for verdadeira, nada acontece. Em outras palavras, **assert** se comporta como **pass**. - Se a expressão lógica for falsa, é gerada uma exceção, opcionalmente acompanhada de uma explicação fornecida pelo programador.

Neste caso, por exemplo, poderíamos escrever...

```
In [20]: %reset -f

         def f(n):
             assert isinstance(n, int), \
                 f"o argumento deveria ser do tipo <class 'int'> mas é {type(n)}"
             assert n >= 0, \
                 f"o argumento não pode ser negativo mas é {n}"

             if n > 0:
```

```
            return f(n - 1) * n
        else:
            return 1

print('f(0)  =', f(0))
print('f(1)  =', f(1))
print('f(5)  =', f(5))
print('f(10) =', f(10))
```

```
f(0)  = 1
f(1)  = 1
f(5)  = 120
f(10) = 3628800
```

Veja que nada mudou nos exemplos válidos. Já nos casos inválidos...

```
In [21]: print('f("a")  =', f("a"))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)

<ipython-input-21-6793c4d60eec> in <module>()
----> 1 print('f("a")  =', f("a"))


<ipython-input-20-92be3a760a45> in f(n)
  2
  3 def f(n):
----> 4     assert isinstance(n, int),         f"o argumento deveria ser do tipo <class '
  5     assert n >= 0,         f"o argumento não pode ser negativo mas é {n}"
  6


AssertionError: o argumento deveria ser do tipo <class 'int'> mas é <class 'str'>
```

```
In [22]: print('f(-5)  =', f(-5))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)

<ipython-input-22-b17ddd6c5ea9> in <module>()
----> 1 print('f(-5)  =', f(-5))


<ipython-input-20-92be3a760a45> in f(n)
  3 def f(n):
  4     assert isinstance(n, int),         f"o argumento deveria ser do tipo <class '
```

```
----> 5     assert n >= 0,         f"o argumento não pode ser negativo mas é {n}"
      6
      7     if n > 0:


AssertionError: o argumento não pode ser negativo mas é -5
```

O uso de asserções para validar argumentos muitas vezes é considerado inadequado porque interrompe drasticamente a execução do programa, impedindo qualquer ação corretiva por parte do usuário.
Uma outra possibilidade seria retornar um resultado indicativo de que algo de estranho aconteceu, p.ex. *None*.

```
In [23]: %reset -f

         def f(n):
             if not isinstance(n, int) or n < 0:
                 return None
             elif n > 0:
                 return f(n - 1) * n
             else:
                 return 1

         print('f(0)   =', f(0))
         print('f(1)   =', f(1))
         print('f(5)   =', f(5))
         print('f(10)  =', f(10))
         print('f(-5)  =', f(-5))
         print('f("a") =', f("a"))
```

```
f(0)   = 1
f(1)   = 1
f(5)   = 120
f(10)  = 3628800
f(-5)  = None
f("a") = None
```

Desenvolvimento de uma solução não-recursiva
Uma definição recursiva como essa é frequentemente transformada numa ***definição recorrente*** por razões de eficiência.
Para isso, note que a expressão recursiva pode também ser vista como a definição de uma sequência de produtos
$p_0 = 1$
$p_1 = 1$
$p_2 = 1 \times 2$
...
$p_n = 1 \times 2 \times \ldots \times n$
Portanto, $p_n = p_{n-1} \times n$, isto é, resulta da acumulação de um produto, o que pode ser conseguido com um comando iterativo simples. Por exemplo,

```
p = 1
for i in range (2, n + 1):
    p = p * i
```

Esse modelo ajusta-se naturalmente ao caso em que $n = 0$ como mostra a implementação abaixo.

```
In [24]: %reset -f

         def f(n):
             if not isinstance(n, int) or n < 0:
                 return None
             else:
                 p = 1
                 for i in range (2, n + 1):
                     p = p * i
                 return p

         print('f(0)   =', f(0))
         print('f(1)   =', f(1))
         print('f(5)   =', f(5))
         print('f(10)  =', f(10))
         print('f(-5)  =', f(-5))
         print('f("a") =', f("a"))
```

```
f(0)   = 1
f(1)   = 1
f(5)   = 120
f(10)  = 3628800
f(-5)  = None
f("a") = None
```

Podemos usar asserções também para "encapsular" nossos testes e, assim, deixá-los no corpo do programa permanentemente sem que eles interfiram na execução.
É claro que, para poder testar nossa função, é preciso ser capaz de antecipar a resposta correta em um certo número de casos.
Procure sempre antecipar os casos críticos, por exemplo valores extremos dos dados ou pontos em que se espera alguma mudança no comportamento da função.

```
In [25]: %reset -f

         def f(n):
             if not isinstance(n, int) or n < 0:
                 return None
             else:
                 p = 1
                 for i in range (1, n + 1):
                     p = p * i
                 return p

         tst = [(0, 1), (1, 1), (5, 120), (10, 3628800), (-5, None), ('abc', None)]
```

```
        for a, r in tst:
            assert f(a) == r, \
                f'f({a}) = {f(a)} (valor esperado: {r})'
```

1.7 Uma função pode definir variáveis locais

Uma variável definida dentro de uma função é dita *local*. Ela existe apenas durante a execução da chamada e não é visível fora da função.

O que aconteceria se o programa principal também tivesse uma variável com o mesmo nome?

A variável local "ocultaria" a variável externa durante a execução da chamada, como mostra o exemplo abaixo, onde o programa chama a função f que chama a função g e todos definem uma variável v.

```
In [26]: %reset -f

         def g():
             v = 'sou v e fui definida na função g'
             print('g():', v)

         def f():
             v = 'sou v e fui definida na função f'
             g()
             print('f():', v)

         v = 'sou v e fui definida no programa principal'
         f()
         print('pp: ', v)
```

```
g(): sou v e fui definida na função g
f(): sou v e fui definida na função f
pp:  sou v e fui definida no programa principal
```

Você consegue dizer em que sequência os comandos do script acima foram executados? - 1, 3, 7, 12, 13, 8, 9, 4, 5, 10, 14

A execução de um *def* apenas define o nome e o cabeçalho da função, habilitando a sua chamada em comandos posteriores.

O corpo da função só será executado posteriormente em resposta a essas chamadas.

Nesse exemplo, cada nova definição do nome v oculta a definição anterior existente no mesmo espaço.

O que acontece se o corpo de uma função se referir a uma variável que não está definida localmente?

Se a variável estiver definida no espaço que "contém" a função, essa definição será usada. Caso contrário, será gerada uma exceção.

Por exemplo, vamos "comentar" a linha 4 e, com isso, remover a definição de v na função g.

```
In [27]: %reset -f

         def g():
```

```
            # v = 'sou v e fui definida na função g'
            print('g():', v)

        def f():
            v = 'sou v e fui definida na função f'
            g()
            print('f():', v)

        v = 'sou v e fui definida no programa principal'
        f()
        print('pp: ', v)
```

```
g(): sou v e fui definida no programa principal
f(): sou v e fui definida na função f
pp:  sou v e fui definida no programa principal
```

Como v não está definida em g, foi usada a definição existente no espaço onde está contida a definição de g, ou seja, o programa principal.
Note que, se comentarmos também a linha 12, o programa passará a gerar uma exceção, uma vez que a definição de v dentro de f não é visível em g (porque f não "contém" g).

```
In [28]: %reset -f
```

```
        def g():
            # v = 'sou v e fui definida na função g'
            print('g():', v)

        def f():
            v = 'sou v e fui definida na função f'
            g()
            print('f():', v)

        # v = 'sou v e fui definida no programa principal'
        f()
        print('pp: ', v)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-28-d6d837ffaf8a> in <module>()
     11
     12 # v = 'sou v e fui definida no programa principal'
---> 13 f()
     14 print('pp: ', v)


<ipython-input-28-d6d837ffaf8a> in f()
      7 def f():
      8     v = 'sou v e fui definida na função f'
```

```
----> 9     g()
     10     print('f():', v)
     11

<ipython-input-28-d6d837ffaf8a> in g()
      3 def g():
      4     # v = 'sou v e fui definida na função g'
----> 5     print('g():', v)
      6
      7 def f():

NameError: name 'v' is not defined
```

1.7.1 Exemplo: Verificar se uma cadeia de caracteres é palíndroma

Criar uma função que diga se uma cadeia de caracteres é palíndroma, ignorando acentos, maiúsculas e pontuação.

Vamos começar esboçando uma solução...

```
In [29]: def é_palíndroma(s):
             s = sem_acentos(s)
             s = s.lower()
             s = só_letras(s)
             return é_pal(s)
```

Vamos definir as funções auxiliares...

```
In [30]: def sem_acentos(s):
             acentos = {'á': 'a', 'à': 'a', 'ã': 'a', 'â': 'a',
                        'é': 'e', 'ê': 'e',
                        'í': 'i',
                        'ó': 'o', 'õ': 'o', 'ô': 'o',
                        'ú': 'u',
                        'ç': 'c'
                       }
             rs = ''
             for c in s:
                 rs = rs + acentos.get(c, c)
             return rs

In [31]: def só_letras(s):
             minúsculas = 'abcdefghijklmnopqrstuvwxyz'
             rs = ''
             for c in s:
                 if c in minúsculas:
                     rs = rs + c
             return rs
```

E agora vamos definir a função local...

```
In [32]: def é_palíndroma(s):
             def é_pal(s):
                 return (len(s) == 0) or (s[0] == s[-1] and é_pal(s[1:-1]))

             s = sem_acentos(s)
             s = s.lower()
             s = só_letras(s)
             return é_pal(s)

         print(é_palíndroma('Socorram-me subi no ônibus em Marrocos.'))
```

```
True
```

1.8 Mais exemplos...

1.8.1 Calcular o maior de três números

Criar uma função *maior_de_3*(a, b, c) que calcule e retorne o valor do maior de seus três parâmetros.

Desenvolvimento

Ao procurar uma solução para um problema, é quase sempre útil começar com uma versão mais restrita em algum aspecto que nos permita exercitar o raciocínio num espaço mais limitado.
Frequentemente, a solução do problema real acaba sendo uma expansão dessa solução particular.
Neste caso, por exemplo, poderíamos pensar em criar uma função *maior_de_2*(a, b) que calcula e retorna o valor do maior de seus dois parâmetros.

```
In [33]: %reset -f

         def maior_de_2(a, b):
             if a > b:
                 return a
             else:
                 return b

         tst = [(0, 0, 0), (1, 0, 1), (5, 1, 5), (-9, 1, 1)]

         for a, b, r in tst:
             assert maior_de_2(a, b) == r, \
                 f'maior_de_2({a}, {b}) = {maior_de_2(a, b)} (valor esperado: {r})'
```

Tendo criado e testado a função *maior_de_2*, fica fácil criar *maior_de_3* ...

```
In [34]: %reset -f

         def maior_de_2(a, b):
             if a > b:
                 return a
             else:
                 return b
```

```
        def maior_de_3(a, b, c):
            if maior_de_2(a, b) == a:
                return maior_de_2(a, c)
            else:
                return maior_de_2(b, c)

        tst = [(0, 0, 0, 0), (1, 0, 0, 1), (0, 1, 0, 1),
               (0, 0, 1, 1), (1, 2, 3, 3), (1, 5, -9, 5)]

        for (a, b, c, r) in tst:
            assert maior_de_3(a, b, c) == r, \
                f'maior_de_3({a}, {b}, {c}) = {maior_de_3(a, b, c)} (valor esperado: {r})'
```

Essas funções também poderiam ter sido escritas de forma ligeiramente mais compacta...

```
In [35]: %reset -f

        def maior_de_2(a, b):
            return a if a > b else b

        def maior_de_3(a, b, c):
            return maior_de_2(a, c) if maior_de_2(a, b) == a else maior_de_2(b, c)

        tst = [(0, 0, 0, 0), (1, 0, 0, 1), (0, 1, 0, 1),
               (0, 0, 1, 1), (1, 2, 3, 3), (1, 5, -9, 5)]

        for (a, b, c, r) in tst:
            assert maior_de_3(a, b, c) == r, \
                f'maior_de_3({a}, {b}, {c}) = {maior_de_3(a, b, c)} (valor esperado: {r})'
```

1.8.2 Inverter uma cadeia de caracteres

Criar uma função $inv\_cadeia(s)$ que inverta uma cadeia de caracteres passada como argumento.

Você consegue pensar em alguns casos particulares cuja solução seria trivial, ou pelo menos mais simples?

Por exemplo, $inv\_cadeia(s) = s$, se $len(s) \leq 1$.

Você consegue escrever uma definição de $inv\_cadeia(s)$, a partir de um caso mais simples?

Por exemplo, $inv\_cadeia(s) = inv\_cadeia(s[1:]) + s[0]$.

Essas duas definições são suficientes para que desenvolvamos uma versão recursiva de $inv\_cadeia(s)$.

```
In [36]: def inv_cadeia(s):
            if s:
                return inv_cadeia(s[1:]) + s[0]
            else:
                return s

        la = ['123abc', '', 'x', 'anilina']
        lr = ['cba321', '', 'x', 'anilina']
        for a, r in zip(la, lr):
            assert inv_cadeia(a) == r, \
                f"inv_cadeia({a}) = {inv_cadeia(a)} (valor esperado: {r})"
```

Também é possível pensar numa versão recorrente (ou iterativa) de *inv_cadeia*(s).

Num algoritmo recorrente, geralmente buscamos transformar o estado de um objeto de um valor inicial conhecido em um valor final desejado, tratando um componente desse objeto por iteração.

Neste caso, basta pegar os caracteres da cadeia de entrada s um-a-um e colocá-los na posição apropriada da cadeia de saída *invs*.

Como desejamos que *invs* seja o inverso de s, cada caractere deverá ser colocado nela na posição oposta à em que ele se encontrava na cadeia original.

Uma maneira simples de implementar essa solução é usando um comando *for*:

```
for c in s:
    # colocar c na posição adequada de invs
```

Como `for` retira `c` do início da cadeia `s`, `c` deverá ser colocado no final da cadeia `invs`.

Por exemplo, suponha que `s == '12345'` e que já tenham sido processados os três primeiros caracteres.
Nesse momento `c == '4'` e `invs == '321'`.
Para completar corretamente a iteração devemos colocar `c` no início da cadeia `invs` e, para isso, executar `invs = c + sinv`.

No início, a cadeia `invs` deverá estar vazia.

```
In [37]: def inv_cadeia(s):
             invs = ''
             for c in s:
                 invs = c + invs
             return invs

         la = ['123abc', '', 'x', 'anilina']
         lr = ['cba321', '', 'x', 'anilina']
         for a, r in zip(la, lr):
             assert inv_cadeia(a) == r, \
                 f'inv_cad({a}) = {inv_cad(a)} (valor esperado: {r})'
```

1.8.3 Ler uma lista e retornar outra sem os elementos repetidos na primeira

Neste caso basta percorrer a lista original, extraindo seus itens um-a-um.
Se o item extraído ainda não estiver na lista de saída, ele é colocado lá.
Caso contrário, ele é ignorado.

```
In [38]: %reset -f

         def vals_únicos(lista):
             assert isinstance(lista, list), \
                 f"tipo do argumento = {type(lista)}, deveria ser <class 'list'>"

             unilista = []
             for x in lista:
                 if x not in unilista:
                     unilista += [x]
             return unilista

         assert vals_únicos([]) == []
```

```
        assert vals_únicos([1]) == [1]
        assert vals_únicos([1, 1, 1, 1, 1]) == [1]
        assert vals_únicos([1, 2, 3, 4, 5]) == [1, 2, 3, 4, 5]
        assert vals_únicos([1, 2, 1, 2, 1, 2]) == [1, 2]

        print('todos os testes ok...')
```

```
todos os testes ok...
```

Esta solução também poderia ser "compactada" usando-se uma *list comprehension* que inclui na lista de saída todos os itens da lista de entrada que ***ainda*** não tiverem aparecido.

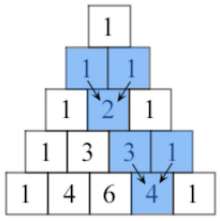
```
In [39]: %reset -f
```

```
        def vals_únicos(lista):
            assert isinstance(lista, list), \
                f"tipo do argumento = {type(lista)}, " +  \
                f"deveria ser <class 'list'>"

            return [x for i, x in enumerate(lista)
                      if x not in lista[:i]]

        assert vals_únicos([]) == []
        assert vals_únicos([1]) == [1]
        assert vals_únicos([1, 1, 1, 1, 1]) == [1]
        assert vals_únicos([1, 2, 3, 4, 5]) == [1, 2, 3, 4, 5]
        assert vals_únicos([1, 2, 1, 2, 1, 2]) == [1, 2]

        print('todos os testes ok...')
```

```
todos os testes ok...
```

1.8.4 Exibir as n primeiras linhas do "triângulo de Pascal"

O "triângulo de Pascal" é uma figura geométrica e aritmética concebida por Blaise Pascal, em que cada número é igual à soma dos dois números que ficam imediatamente acima dele:



Você consegue escrever a expressão que gera os itens de uma linha genérica do triângulo? Experimente começar com um caso particular. Por exemplo, o quarto item da linha 5 é formado pela soma do terceiro e quarto itens da linha 4. Podemos conseguir esse efeito somando duas cópias da linha 4, deslocadas de uma coluna, ...

linha 4	1	3	3	1	
linha 4 *deslocada*		1	3	3	1
linha 5	1	4	6	4	1

Suponha que a linha 4 esteja representada numa lista *tp*.
Podemos ajustar o comprimento das duas linhas anexando uma lista $[0]$ nas posições adequadas.
A primeira linha da tabela será dada por $tp + [0]$.
A segunda linha da tabela, por sua vez, será dada por $[0] + tp$.
Para "emparelhar" essas duas linhas podemos usar a função *zip* e depois somar os itens correspondentes usando um *for*.

```
In [40]: def triângulo_de_pascal(n):

             def próxima_linha(tp):
                 if tp:
                     ptp = []
                     for esq, dir in zip(tp + [0], [0] + tp):
                         ptp += [esq + dir]
                     return ptp
                 else:
                     return [1]

             if not isinstance(n, int) or n < 1:
                 return None
             else:
                 tp = []
                 for linha in range(n):
                     tp = próxima_linha(tp)
                     print(tp)

         triângulo_de_pascal(6)
```

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
```

Finalmente, é possível compactar a função *prxima_linha* usando uma *list comprehension*.

```
In [41]: def triângulo_de_pascal(n):

             def próxima_linha(tp):
                 if tp:
                     return [esq + dir for esq, dir in zip(tp + [0], [0] + tp)]
                 else:
                     return [1]

             if not isinstance(n, int) or n < 1:
                 return None
             else:
                 tp = []
                 for linha in range(n):
```

```
                    tp = próxima_linha(tp)
                    print(tp)

        triângulo_de_pascal(6)
```

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
```