


Módulo de Programação Python: Introdução à Linguagem

Terceiro Encontro

Apresentação de Python: Modularização de código, Funções.

 No description has been provided for this image

Objetivo: Introduzir a técnica de modularização de código com ajuda de funções. Ensinar iteração avançada e compreensões de lista e dicionário.

Funções, como utilizar e implementar

Algoritmos simples podem ser implementados utilizando apenas variáveis e estruturas de controle de fluxo. O desenvolvimento de programas e códigos mais elaborados, exigem a utilização de funções, inclusive daquelas que aparecem na forma de métodos de classes. Paradigmas de programação como o de programação estruturada ou programação orientada a objetos, que visam entre outras coisas, podermos reutilizar códigos e estruturar os mesmo de forma a simplificar seu desenvolvimento e posterior manutenção, nos levam à necessidade de aprendermos a implementar funções de forma eficiente.

Que é uma função

Uma função não é mais que um bloco de código ou rotina, que pode ser utilizado múltiplas vezes e que permite estruturar o código de forma a garantir uma sequencia mais limpa e legível de instruções. O bloco sintático vinculado à função é sempre associado a um nome que serve para invocar a execução do mesmo a qualquer momento. Como na maioria das linguagens tradicionais, em **Python**, a chamada a uma funções evolve seu nome seguido de parêntesis que delimitam os parâmetros que deverão ser passados para a função. Mesmo funções que não recebem parâmetros são chamadas utilizando parêntesis. Os parêntesis ajudam a distinguir quando se esta fazendo referenciando uma variável ou fazendo a chamada a uma função.

Semelhante ao conceito matemático de função, seu equivalente computacional pode ser utilizar para mapear um conjunto de entrada, ou domínio, em um conjunto de saída ou imagem. Entretanto as funções, computacionalmente falando, podem ter domínio vazio, não recebem parâmetros de entrada, e também podem ter imagem nula, não retornam nenhum resultado.

Até aqui foram utilizados, em vários momentos, funções como o caso de:

```
In [1]: print("Olha uma função aqui!!!")
```

Olha uma função aqui!!!

Vamos aprender então como criar nossas próprias funções em **Python**.

Definindo funções

Para se definir uma função em **Python** utilizamos a palavra chave `def`, seguido do nome da função e, entre parêntesis, os parâmetros de entrada. O bloco de instruções associado a aquela

função pode ser declarado, seguindo a sintaxe apropriada em **Python**, ou seja utilizar **:** e indentação para delimitar o bloco de instruções associada à mesma. Uma função em **Python** é, então, um objeto que engloba um bloco sintático definido através da palavra chave **def** com a seguinte sintaxe:

```
In [2]: # Uma função ainda não implementada
def minhaFunção():
    pass

# A função pode ser utilizada
minhaFunção()
print(type(minhaFunção))
print(type(minhaFunção()))
```

```
<class 'function'>
<class 'NoneType'>
```

A `minhaFunção` está definida de forma que ela não recebe parâmetros nem faz absolutamente nada. A palavra chave `pass` pode ser utilizada quando queremos deixar definido um bloco de código para o qual ainda não temos uma implementação apropriada.

```
In [3]: #Função sem argumentos. Retorna uma string
def funSemArgumentos():
    # Bloco sintático
    saída = "Esta função não tem argumentos"
    #saída = 3
    #saída = 3.14
    #saída = True
    #saída = [1,2,3]
    return saída

print(funSemArgumentos())
print(type(funSemArgumentos))
print(type(funSemArgumentos()))
```

```
Esta função não tem argumentos
<class 'function'>
<class 'str'>
```

```
In [4]: #Função com um argumento. Não retorna nada
def funUmArgumento(arg1):
    # Bloco sintático
    arg1.append(5) # Colocando um novo elemento o final da lista

L = []
print(funUmArgumento(L))
print(L)
print(type(funUmArgumento))
print(type(funUmArgumento(L)))
```

```
None
[5]
<class 'function'>
<class 'NoneType'>
```

```
In [5]: #Função com dois argumentos. Retorna a adição dos mesmos
def funDoisArgumentos(arg1, arg2):
    # Bloco sintático
    valor = arg1 + arg2
    return valor

print("2 + 2 = ", funDoisArgumentos(2,2))
print(funDoisArgumentos("a + ", "b = c"))
```

```
print(type(funDoisArgumentos(2,2)))
print(type(funDoisArgumentos(2,2)))
```

```
2 + 2 = 4
a + b = c
<class 'function'>
<class 'int'>
```

Vamos tentar outro exemplo de função mais ilustrativo.

```
In [6]: # Retorna os N primeiros termos da Sequência de Fibonacci
def fibonacci(N):
    L = [0]
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

Temos agora uma função que implementa um algoritmo que permite gerar N termos consecutivos da sequência de Fibonacci. Não lembra de como se calculam os termos da sequência? Veja esta [referência](#)

Podemos utilizar a função da seguinte forma:

```
In [7]: fibonacci(10)
```

```
Out[7]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Reparem que na definição da função, ao contrário de linguagens mais tradicionais como **C/C++**, não estão incluídos o tipo de retorno da função ou os tipos dos parâmetros de entrada da mesma. Isto faz das funções em **Python** um instrumento muito versátil, capaz de retornar valores diversos. O mecanismo de passagem de parâmetros também é bastante inovador, mas sobre ele falaremos mais para frente.

Vejam o exemplo a seguir que mostra como retornar múltiplos resultados:

```
In [8]: def real_imag_conj(val):
        return val.real, val.imag, val.conjugate()
tripla = real_imag_conj(3 + 4j)
print(tripla)
r, i, c = real_imag_conj(3 + 4j)
print(r, i, c)
```

```
(3.0, 4.0, (3-4j))
3.0 4.0 (3-4j)
```

Vejamos outro exemplo. A função no próximo exemplo recebe dois parâmetros, sem especificar o tipo, e utiliza o operador adição com eles para gerar o valor de saída.

```
In [9]: def soma(a, b):
        return a + b
```

Consideremos, entretanto, que o operador adição está definido para diferentes tipos de dados, e para cada tipo ele funciona de uma forma diferente. Esta função pode trabalhar então com qualquer variável em **Python** para a qual esteja definido o operador adição.

```
In [10]: # com tipos numéricos funciona como adição
print(soma(2, 2))
print(soma(2.0, 2))
print(soma(2.0+3j, 2.0))
```

```
# com strings e listas funciona como concatenação
print(soma("dois + ", "dois = quatro"))
print(soma([1,2,3], [4,5,6]))
```

```
4
4.0
(4+3j)
dois + dois = quatro
[1, 2, 3, 4, 5, 6]
```

Isto significa que as funções em **Python** são, essencialmente, polimórficas. Isto é, elas se comportam de acordo com o tipo dos objetos com que estão trabalhando. Vejamos este outro exemplo.

```
In [11]: def intersect(seq1, seq2):
        res = []
        for x in seq1:
            if x in seq2:
                res.append(x)
        return res

print(intersect("Modelagem", "viagem"))
print(intersect([1, 2, 3, 5, 7], [1, 2, 3, 4, 5, 6, 7]))

['e', 'a', 'g', 'e', 'm']
[1, 2, 3, 5, 7]
```

As funções declaradas em **Python** elas também são objetos. Entretanto elas somente viram uma instância específica da classe `function`, apenas quando são executadas. Como qualquer outra declaração, `def` pode ser utilizada nos contextos em que declarações são aceitas. Veja o seguinte exemplo em que declaramos uma nova função utilizando uma estrutura condicional `if-else`.

```
In [12]: #cond = True
        cond = False

        if cond:
            def minhaSoma(a, b):
                return a + b
        else:
            def minhaSoma(a, b):
                return (1/a + 1/b)

        print (minhaSoma(2, 2))
```

```
1.0
```

A palavra chave `def` cria um objeto, e “atribui” uma referência para o mesmo a uma variável que é o nome da função. Isto abre a possibilidade de utilizar nomes diferentes para uma mesma função. Veja o exemplo a seguir.


```
In [13]: adição = soma
        print(adição(2, 2))
        print(adição("dois + ", "dois = quatro"))
        print(type(adição))
        print(type(adição(2,2)))
        print(type(adição("dois + ", "dois = quatro")))
```

```
4
dois + dois = quatro
<class 'function'>
<class 'int'>
<class 'str'>
```

Escopo das variáveis

Um aspecto importante a ser analisado na hora de implementar funções é o escopo das variáveis. Até aqui não prestamos muita atenção a este aspecto. Assumimos que as variáveis declaradas dentro de uma função são variáveis locais. De forma geral em **Python** o escopo de uma variável depende do local onde ela é declarada. Desta forma uma variável pode ser:

- **local** : quando declarada dentro de uma função;
- **non local** : quando declarada dentro de uma função mas antes de uma outra função aninhada dentro dela;
- **global** : quando declarada fora de todas as funções

No description has been provided for this image

(*) Fonte: **Learning Python. 5th Edition. Mark Lutz .**

Veja os exemplos a seguir

```
In [14]: # Escopo global
x = 99 # Aqui x é uma variável global

print("x Global = ", x)

def funçãoX(y): # y é um parâmetro, uma variável Local da função
    #Escopo local
    print("y Local = ", y)
    print("x Global = ", x)
    # z é atribuída dentro do corpo da função (local)
    z = x + y # já x, que não foi definido neste bloco, se refere à variável Global
    print("z Local = ", z)
    return z

# Chamando a função
print("função(1) --> ", funçãoX(1))
# y e z não estão definidas fora da função
try:
    print("z Local = ", z)
except:
    print("Fora da função não é possível acessar o seu escopo local !!!: ")
# x é uma variável global
print("x Global = ", x)
```

```
x Global = 99
y Local = 1
x Global = 99
z Local = 100
função(1) --> 100
Fora da função não é possível acessar o seu escopo local !!!:
x Global = 99
```

```
In [15]: # Escopo global
x = 99 # Aqui x é uma variável global

def funçãoX():
    #Escopo local
    x = 11 # se cria uma nova referência com o nome x, no escopo local
    print("x Local = ", x)

print("x Global = ", x)
print("Chamando à funçãoX()")
```

```
funçãoX()
print("Mas x Global continua= ", x)
```

```
x Global = 99
Chamando à funçãoX()
x Local = 11
Mas x Global continua= 99
```

```
In [16]: # Escopo global
x = 99

def funçãoX():
    #Escopo local
    x = 11

    def funçãoY():
        #Escopo local
        x = 22
        print("x Local da funçãoY: ", x)

    print("x Local da funçãoX: ", x)
    print("Chamando à funçãoY()")
    funçãoY()
    print("Mas x Local da funçãoX continua = ", x)

print("x Global = ", x)
print("Chamando à funçãoX()")
funçãoX()
print("E x Global continua = ", x)
```

```
x Global = 99
Chamando à funçãoX()
x Local da funçãoX: 11
Chamando à funçãoY()
x Local da funçãoY: 22
Mas x Local da funçãoX continua = 11
E x Global continua = 99
```

```
In [17]: # Escopo global
x = 99

def funçãoX():
    #Escopo local
    x = 11

    def funçãoY():
        #Escopo local
        y = 22
        print("y Local da funçãoY: ", y)
        print("x na funçãoY é Nonlocal: ", x)

    print("x Local da funçãoX: ", x)
    print("Chamando à funçãoY()")
    funçãoY()

print("x Global = ", x)
print("Chamando à funçãoX()")
funçãoX()
print("E x Global continua = ", x)
```

```
x Global = 99
Chamando à funçãoX()
x Local da funçãoX: 11
Chamando à funçãoY()
y Local da funçãoY: 22
x na funçãoY é Nonlocal: 11
E x Global continua = 99
```

O escopo de uma variável pode ser modificado utilizando as palavras chaves `global` e `nonlocal`. Veja uma pequena variação do exemplo anterior.

```
In [18]: # Escopo global
x = 99
y = 3
print("x Global = ", x)
print("y Global = ", y)

def funçãoX():
    #Escopo global
    # quando me refira a x dentro, da funçãoX, ...
    global x # estou falando do x global
    x = 11
    print("x Global dentro da funçãoX agora é = ", x)
    # já este y é local da função, ...
    y = 99 # diferente do y de escopo global
    print("y Local dentro da funçãoX = ", y)
    def funçãoY():
        #Escopo nonlocal
        # quando me refira a y dentro, da funçãoY, ...
        nonlocal y # estou falando do y local da funçãoX
        print("y NonLocal dentro da funçãoY = ", y)
        y = 22

    print("Chamando à funçãoY()")
    funçãoY()
    print("Agora y Local da funçãoY() é = ", y)

print("Chamando à funçãoX()")
funçãoX()
print("Agora x Global é = ", x)
print("E y Global é = ", y)
```

```
x Global = 99
y Global = 3
Chamando à funçãoX()
x Global dentro da funçãoX agora é = 11
y Local dentro da funçãoX = 99
Chamando à funçãoY()
y NonLocal dentro da funçãoY = 99
Agora y Local da funçãoY() é = 22
Agora x Global é = 11
E y Global é = 3
```

Passagem de parâmetros

A passagem de parâmetros para funções pode ser feita, de forma geral, de duas formas: por valor ou por referência.

- Na passagem de parâmetros por valor, cria-se dentro da função, uma cópia da variável original de forma que, alterações feitas dentro da função não afetam o valor originalmente armazenado.
- Na passagem por referência se trabalha, dentro da função, no endereço da variável de origem. Por este motivo, modificações feitas neste contexto afetam o valor da variável fora da função.

Em **Python** a passagem de parâmetros é feita da seguinte forma:

- São atribuídos referências a objetos para variáveis locais (parâmetros);
- Fazer novas atribuições a estes parâmetros dentro da função não afeta as variáveis originais;
- Modificar objetos mutáveis referenciados por parâmetros da função afeta as variáveis originais

Isto significa que em **Python**, ainda que na realidade a passagem é sempre feita por referência, na prática:

- A passagem de objetos mutáveis funciona como quando feita por referência
- A passagem de objetos imutáveis funciona como quando feita por valor

Veja como funciona.

```
In [19]: def testePassagemI(x):
          print("x chego como = ", x, type(x))
          x = None
          print("Agora x = ", x, type(x))

          def testePassagemM(x):
              print("x chego como = ", x, type(x))
              x[0] = 0
              print("Agora x = ", x, type(x))

          print("Chamando à função testePassagem()")
          print("Objeto imutável: int")
          a = 3
          testePassagemI(a)
          print("Agora a = ", a, type(a))

          print("Objeto mutável: list")
          a = [1,2,3]
          testePassagemM(a)
          print("Agora a = ", a, type(a))
```

```
Chamando à função testePassagem()
Objeto imutável: int
x chego como =  3 <class 'int'>
Agora x =  None <class 'NoneType'>
Agora a =  3 <class 'int'>
Objeto mutável: list
x chego como =  [1, 2, 3] <class 'list'>
Agora x =  [0, 2, 3] <class 'list'>
Agora a =  [0, 2, 3] <class 'list'>
```

```
In [20]: def funTesteIndex(a):
          try:
              a[0] = "M" # Listas são sempre objetos mutáveis
          except:
              pass

          b = [1, 2, 3]
          print(b)
          funTesteIndex(b)
          print(b)
          c = (1, 2, 3)
          print(c)
          print(c[0])
          funTesteIndex(c)
          print(c)
          d = "mudança"
          print(d)
          print(d[0])
          funTesteIndex(d)
          print(d)
```



```
[1, 2, 3]
['M', 2, 3]
(1, 2, 3)
1
(1, 2, 3)
mudança
m
mudança
```

No cabeçalho de uma função fica definido os nomes dos parâmetros que a função recebe e a ordem em que eles devem ser enviados. **Python** estabelece alguns mecanismos que permitem criar chamadas a funções muito mais flexíveis. Além do mecanismo tradicional (posicional) pode-se destacar outros dois:

- palavras chaves
- valores predefinidos

Veja o seguinte exemplo que demonstra como utilizar o conceito de palavras chaves na passagem de parâmetros.

```
In [21]: # A funçãoY tem três parâmetros de entrada. Pela sua ordem eles são a, b e c
def funçãoY(a, b, c):
    print(a, b, c)

# Posso chamar colocando apenas os valores pela ordem
funçãoY(1, 2, 3) # isto significa que a recebe 1, b recebe 2 e c recebe 3
# Posso especificar o valor que cada parâmetro vai receber
funçãoY(c = 4, a = 7, b = 1) # isto significa que a recebe 7 b recebe 1 e c recebe 4
# Posso utilizar parcialmente a ordem e os nomes
funçãoY(4, c = 2, b = 5) # isto significa que a recebe 4, b recebe 5 e c recebe 2

1 2 3
7 1 4
4 5 2
```

Desta forma **Python** permite que você seja mais específico na hora de chamar a função. Identificar o que você está passando para uma função deixa seu código bem mais interessante e fácil de ler. Mas como saber quais parâmetros uma função recebe e qual a sua ordem? Veja [aqui](#) e [aqui](#) como documentar funções.

```
In [22]: help(len)
```

Help on built-in function len in module builtins:

```
len(obj, /)
    Return the number of items in a container.
```

Uma pequena modificação na definição da funçãoY pode deixar ela mais simples de usar.

```
In [23]: def funçãoY(a = 1, b = 2, c = 3):
...:     """
...:     A funçãoY tem três parâmetros de entrada.
...:     a: com valor padrão 1
...:     b: com valor padrão 2
...:     c: com valor padrão 3
...:
...:     """
...:     print(a, b, c)
```

```
In [24]: help(funçãoY)
```

Help on function funçãoY in module __main__:

funçãoY(a=1, b=2, c=3)

A funçãoY tem três parâmetros de entrada.

a: com valor padrão 1

b: com valor padrão 2

c: com valor padrão 3

```
In [25]: # Posso chamar a função com os valores implícitos das variáveis
funçãoY() # os parâmetros tem seus valores padrão
# Posso chamar a função passando valores explicitamente, pela ordem
funçãoY(4, 5, 6) # isto significa que a recebe 4, b recebe 5 e c recebe 6
# Posso chamar a função passando valores explicitamente, pelo nome
funçãoY(c = 6, a = 4, b = 5) # isto significa que a recebe 4, b recebe 5 e c recebe 6
# Posso chamar a função passando valores explicitamente, pela ordem e pelo nome
funçãoY(1, c = 6, b = 5) # isto significa que a recebe 1, b recebe 5 e c recebe 6
# Posso chamar a função passando valores explicitamente, pela ordem e/ou pelo nome
funçãoY(a=7) # e deixar alguns parâmetros com seus valores implícitos
```

```
1 2 3
4 5 6
4 5 6
1 5 6
7 2 3
```

Python ainda permite função com uma quantidade indeterminada de parâmetros. Com esta finalidade a função deve ser declarada da seguinte forma.

1. O nome do parâmetros precedido de um asterisco (*****) significa que a função espera uma lista de parâmetros;

```
In [26]: def funFlexL(*ListaDeParâmetros):
        print(len(ListaDeParâmetros))
        for index, item in enumerate(ListaDeParâmetros):
            print(f"Parametro {index} - {item}")
        return len(ListaDeParâmetros)

funFlexL(3, 2.4, "teste", [1,2,3], True)
```

```
5
Parametro 0 - 3
Parametro 1 - 2.4
Parametro 2 - teste
Parametro 3 - [1, 2, 3]
Parametro 4 - True
```

Out[26]: 5

A lista pode ter qualquer tamanho.

2. O nome do parâmetros precedido de dois asterisco (******) significa que a função espera um dicionário;

```
In [27]: def funFlexD(**DicionarioDeParâmetros):
        for key, value in DicionarioDeParâmetros.items():
            print(f"Parâmetro {key} - {value}")

funFlexD(a = 1, beta = 'a', y = 2.0, L = [1,3])
```

```
Parâmetro a - 1
Parâmetro beta - a
Parâmetro y - 2.0
Parâmetro L - [1, 3]
```

3. A função pode receber ainda uma lista e um dicionário;

```
In [28]: def funFlexLD(*ListaDeParâmetros, **DicionarioDeParâmetros):  
    for index, item in enumerate(ListaDeParâmetros):  
        print(f"Parametro {index} - {item}")  
    for key, value in DicionarioDeParâmetros.items():  
        print(f"Parâmetro {key} - {value}")  
  
funFlexLD(3, 2.4, "teste", [1,2,3], True, a = 1, beta = 'a', y = 2.0, L = [1,3])
```

```
Parametro 0 - 3  
Parametro 1 - 2.4  
Parametro 2 - teste  
Parametro 3 - [1, 2, 3]  
Parametro 4 - True  
Parâmetro a - 1  
Parâmetro beta - a  
Parâmetro y - 2.0  
Parâmetro L - [1, 3]
```

Funções anônimas (lambda)

Além do mecanismo que vimos até aqui para definir funções, utilizando a palavra reservada `def`, é possível definir, em **Python**, outra maneira de definir funções curtas a instrução `lambda`. Veja este exemplo simples:

```
In [29]: troca = lambda x, y: (y, x)  
a = 1  
b = 2  
print("a = ", a, "b = ", b)  
a, b = troca(a,b)  
print("a = ", a, "b = ", b)
```

```
a = 1 b = 2  
a = 2 b = 1
```

Esta função `lambda` é equivalente à função:

```
In [30]: def troca(x, y):  
    return y, x  
a = 1  
b = 2  
print("a = ", a, "b = ", b)  
a, b = troca(a,b)  
print("a = ", a, "b = ", b)
```

```
a = 1 b = 2  
a = 2 b = 1
```

Então, por que você iria querer usar uma função `lambda`? Temos que lembrar que, em **Python** *tudo é um objeto*, até mesmo as próprias funções! Ou seja, isto significa que funções podem ser passadas como argumentos para funções.

Vejam este exemplo:

```
In [31]: data = [{'Nome': 'Thiago', 'Sobrenome': 'Alves Sena', 'NumMatricula': 202311234},  
                {'Nome': 'Bruno', 'Sobrenome': 'Barbosa Santana', 'NumMatricula': 202324352},  
                {'Nome': 'Breno', 'Sobrenome': 'Carvalho Rios', 'NumMatricula': 202224521}]
```

Esta lista tem apenas três itens, poderia ter muito mais. Gostaríamos de ordenar a mesma mas, não temos um método definido para ordenar dicionários. Desta forma:

```
In [32]: try:
        sorted(data)
    except Exception as e:
        print(e)
```

'<' not supported between instances of 'dict' and 'dict'

Vejamos como podemos contornar esta limitação utilizando uma função `lambda`

```
In [33]: nome = lambda x: x['Nome']
        nome(data[0])
```

Out[33]: 'Thiago'

```
In [34]: # Podemos ordenar pelo campo nome
        help(sorted)
```

Help on built-in function sorted in module builtins:

```
sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.
```

A custom key function can be supplied to customize the sort order, and the reverse flag can be set to request the result in descending order.

```
In [35]: sorted(data, key = nome)
```

```
Out[35]: [{'Nome': 'Breno', 'Sobrenome': 'Carvalho Rios', 'NumMatricula': 202224521},
          {'Nome': 'Bruno', 'Sobrenome': 'Barbosa Santana', 'NumMatricula': 202324352},
          {'Nome': 'Thiago', 'Sobrenome': 'Alves Sena', 'NumMatricula': 202311234}]
```

Tratamento de exceções.

Independentemente das habilidades de um desenvolvedor e de sua equipe de colaboradores, erros de programação acontecem de forma rotineira no processo de desenvolvimento de softwares. Os erros podem ser classificados em uma das seguintes categorias:

- *Erros de sintaxe*: Erros onde o código não segue as regras sintáticas de **Python**, ou seja não é um código **Python** válido. Este tipo de erro é facilmente detetável, ainda mais com a ajuda de uma IDE apropriada.
- *Erros de tempo de execução*: Erros em que o código está sintaticamente correto mas acontece uma falha na execução, eventualmente devido a uma entrada inválida do usuário. Este tipo de erro pode ser identificado e previsto.
- *Erros semânticos*: Erros na lógica da programação: o código é executado sem problemas, mas o resultado não é o esperado. Nestes casos o erro é mais difícil de rastrear e corrigir.

Vamos abordar, nesta seção, como tratar os erros de tempo de execução. Aqui vamos nos concentrar em como lidar de forma limpa com *erros de tempo de execução*. Já vimos, nas aulas até aqui, que **Python** permite lidar com este tipo de erros utilizando uma estrutura apropriada para tratamento de exceções.

Gerenciando erros de tempo de execução

Os erros de tempo de execução podem acontecer de diversas maneiras. Alguns exemplos simples podem se dar quando tentamos utilizar uma variável que ainda não foi definida:

```
In [36]: print("Valor da variável novaVar= ", novaVar)
```

```

-----
NameError                                Traceback (most recent call last)
/Users/evalero/Workspace/GitRepos/ResTIC18_PythonBasico/Notebooks/Aula-03.ipynb Célula
65 line 1
----> <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y104sZmIsZQ%3D%3D?line=0'>1</a> print("Valor da variável
novaVar= ", novaVar)

NameError: name 'novaVar' is not defined

```

Ou quando tentamos utilizar uma operação que não está definida

```
In [ ]: novaVar = [1,2,3] + "outra Lista"
```

Ou quiça, o mais conhecido dos erros de tempo de execução, quando tentamos calcular um resultado matematicamente indefinido:

```
In [37]: novaVar = 5 + 3.14 / 0
```

```

-----
ZeroDivisionError                        Traceback (most recent call last)
/Users/evalero/Workspace/GitRepos/ResTIC18_PythonBasico/Notebooks/Aula-03.ipynb Célula
69 line 1
----> <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y111sZmIsZQ%3D%3D?line=0'>1</a> novaVar = 5 + 3.14 / 0

ZeroDivisionError: float division by zero

```

Ja aconteceu também, em alguns exemplos anteriores, quando foi feito um acesso a uma posição de uma lista que não existe.

```
In [38]: novaVar = [1, 2, 3, 4]
print(novaVar[5])
```

```

-----
IndexError                                Traceback (most recent call last)
/Users/evalero/Workspace/GitRepos/ResTIC18_PythonBasico/Notebooks/Aula-03.ipynb Célula
71 line 2
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y114sZmIsZQ%3D%3D?line=0'>1</a> novaVar = [1, 2, 3, 4]
----> <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y114sZmIsZQ%3D%3D?line=1'>2</a> print(novaVar[5])

IndexError: list index out of range

```

Repare que, em cada caso, Python foi capaz de indicar que um erro aconteceu e ainda entregar uma mensagem com informações sobre o que exatamente deu errado, junto com a linha exata de código onde o erro aconteceu. Ter acesso a erros significativos como esse é imensamente útil ao tentar rastrear a raiz dos problemas no seu código.

Capturando exceções: utilizando `try` e `except`

A principal ferramenta que o Python oferece para lidar com exceções de tempo de execução é a cláusula `try ... except`.

Sua estrutura básica e dada da seguinte forma:

```
In [39]: #geraErro = False
        geraErro = True
        try:
            print("Tentar executar isto primeiro ...")
```

```

    if geraErro:
        novaVar = 1/0
except:
    print("Executar isto apenas se aconteceu um erro !!!")

```

Tentar executar isto primeiro ...

Executar isto apenas se aconteceu um erro !!!

Vamos entender melhor como esta estrutura funciona utilizando este exemplo de divisão por zero.

Suponha que implementamos a seguinte função.

```

In [40]: def divNum(a, b):
        return a/b

# Funciona para valores apropriados:
print("4/2 = ", divNum(4, 2))
# Mas gera um erro de execução se utilizamos uma divisão por zero:
print("4/0 = ", divNum(4, 0))

```

4/2 = 2.0

```

-----
ZeroDivisionError                                Traceback (most recent call last)
/Users/evalero/Workspace/GitRepos/ResTIC18_PythonBasico/Notebooks/Aula-03.ipynb Célula
76 line 7
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y122sZm1sZQ%3D%3D?line=4'>5</a> print("4/2 = ", divNum
(4, 2))
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y122sZm1sZQ%3D%3D?line=5'>6</a> # Mas gera um erro de ex
ecução se utilizamos uma divisão por zero:
----> <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y122sZm1sZQ%3D%3D?line=6'>7</a> print("4/0 = ", divNum
(4, 0))

/Users/evalero/Workspace/GitRepos/ResTIC18_PythonBasico/Notebooks/Aula-03.ipynb Célula
76 line 2
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y122sZm1sZQ%3D%3D?line=0'>1</a> def divNum(a, b):
----> <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y122sZm1sZQ%3D%3D?line=1'>2</a>     return a/b

ZeroDivisionError: division by zero

```

```

In [41]: # Podemos melhorar a implementação utilizando um tratamento de exceções
def divNum(a, b):
    try:
        return a/b
    except:
        return 1E100 # Um número muito grande pode ser uma aproximação para infinito

# Funciona para valores apropriados:
print("4/2 = ", divNum(4, 2))
# E agora funciona também para divisões por zero:
print("4/0 = ", divNum(4, 0))

```

4/2 = 2.0

4/0 = 1e+100

```

In [43]: # Entretanto ela agora funciona também para outros erros:
print("4/'0' = ", divNum(4, "0"))

```

```

TypeError                                Traceback (most recent call last)
/Users/evalero/Workspace/GitRepos/ResTIC18_PythonBasico/Notebooks/Aula-03.ipynb Célula
78 line 2
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y124sZmIsZQ%3D%3D?line=0'>1</a> # Entretanto ela agora f
unciona também para outros erros:
----> <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y124sZmIsZQ%3D%3D?line=1'>2</a> print("4/'0' = ", divNum
(4, "0"))

/Users/evalero/Workspace/GitRepos/ResTIC18_PythonBasico/Notebooks/Aula-03.ipynb Célula
78 line 3
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y124sZmIsZQ%3D%3D?line=0'>1</a> def divNum(a, b):
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y124sZmIsZQ%3D%3D?line=1'>2</a>     try:
----> <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y124sZmIsZQ%3D%3D?line=2'>3</a>         return a/b
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y124sZmIsZQ%3D%3D?line=3'>4</a>     except ZeroDivisionE
rror:
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y124sZmIsZQ%3D%3D?line=4'>5</a>         return 1E100

TypeError: unsupported operand type(s) for /: 'int' and 'str'

```

Nesta caso específico seria apropriado identificar que o erro de execução não é mais de divisão por zero. Desta forma o tratamento deveria ser diferente. Seria recomendável então capturar qual o tipo de erro que aconteceu, ou seja, qual exceção foi gerada. Veja como fica quando capturamos uma exceção específica.

```

In [42]: def divNum(a, b):
          try:
              return a/b
          except ZeroDivisionError:
              return 1E100 # Um número muito grande pode ser uma aproximação para infinito

          print("4/2 = ", divNum(4, 2))
          print("4/0 = ", divNum(4, 0))
          print("4/'0' = ", divNum(4, "0"))

4/2 = 2.0
4/0 = 1e+100

```



```

TypeError                                Traceback (most recent call last)
/Users/evalero/Workspace/GitRepos/ResTIC18_PythonBasico/Notebooks/Aula-03.ipynb Célula
80 line 9
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y126sZmIsZQ%3D%3D?line=6'>7</a> print("4/2 = ", divNum
(4, 2))
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y126sZmIsZQ%3D%3D?line=7'>8</a> print("4/0 = ", divNum
(4, 0))
----> <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y126sZmIsZQ%3D%3D?line=8'>9</a> print("4/'0' = ", divNum
(4, "0"))

/Users/evalero/Workspace/GitRepos/ResTIC18_PythonBasico/Notebooks/Aula-03.ipynb Célula
80 line 3
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y126sZmIsZQ%3D%3D?line=0'>1</a> def divNum(a, b):
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y126sZmIsZQ%3D%3D?line=1'>2</a>         try:
----> <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y126sZmIsZQ%3D%3D?line=2'>3</a>             return a/b
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y126sZmIsZQ%3D%3D?line=3'>4</a>         except ZeroDivisionE
rror:
      <a href='vscode-notebook-cell:/Users/evalero/Workspace/GitRepos/ResTIC18_PythonB
asico/Notebooks/Aula-03.ipynb#Y126sZmIsZQ%3D%3D?line=4'>5</a>             return 1E100

TypeError: unsupported operand type(s) for /: 'int' and 'str'

```

Lançando Exceções: `raise`

O lançamento de exceções é um recurso importante da linguagem para tratamento de erros de tempo de execução. As informações oferecidas por este tipo de evento permitem tratar de forma apropriada o código e evitar situações específicas que podem ser abordadas de forma mais seguras. Mas como gerar nossas próprias exceções?

Para esta finalidade **Python** disponibiliza a instrução `raise`. Veja como usar

```

In [44]: def divNum(a, b):
          if type(b) == type('zero'):
              raise TypeError("0 divisor não pode ser uma string")
          try:
              return a/b
          except ZeroDivisionError:
              return 1E100 # Um número muito grande pode ser uma aproximação para infinito

          try:
              print("4/2 = ", divNum(4, 2))
              print("4/0 = ", divNum(4, 0))
              print("4/'0' = ", divNum(4, "0"))
          except TypeError:
              print("0 divisor não pode ser uma string")

```

```

4/2 = 2.0
4/0 = 1e+100
0 divisor não pode ser uma string

```

Python fornece uma estrutura hierárquica de classes para exceções ([Veja aqui](#)):

```

BaseException
├── BaseExceptionGroup
├── GeneratorExit

```


- |└─ KeyboardInterrupt
- |└─ SystemExit
- |└─ Exception
- |└─ ArithmeticError
- | |└─ FloatingPointError
- | |└─ OverflowError
- | |└─ ZeroDivisionError
- |└─ AssertionError
- |└─ AttributeError
- |└─ BufferError
- |└─ EOFError
- |└─ ExceptionGroup [BaseExceptionGroup]
- |└─ ImportError
- | |└─ ModuleNotFoundError
- |└─ LookupError
- | |└─ IndexError
- | |└─ KeyError
- |└─ MemoryError
- |└─ NameError
- | |└─ UnboundLocalError
- |└─ OSError
- | |└─ BlockingIOError
- | |└─ ChildProcessError
- | |└─ ConnectionError
- | | |└─ BrokenPipeError
- | | |└─ ConnectionAbortedError
- | | |└─ ConnectionRefusedError
- | | |└─ ConnectionResetError
- | |└─ FileExistsError
- | |└─ FileNotFoundError
- | |└─ InterruptedError
- | |└─ IsADirectoryError
- | |└─ NotADirectoryError
- | |└─ PermissionError
- | |└─ ProcessLookupError
- | |└─ TimeoutError
- |└─ ReferenceError
- |└─ RuntimeError
- | |└─ NotImplementedError
- | |└─ RecursionError
- |└─ StopAsyncIteration
- |└─ StopIteration
- |└─ SyntaxError
- | |└─ IndentationError
- | |└─ TabError
- |└─ SystemError
- |└─ TypeError
- |└─ ValueError
- | |└─ UnicodeError
- | |└─ UnicodeDecodeError

- | └── UnicodeEncodeError
- | └── UnicodeTranslateError
- └── Warning
 - └── BytesWarning
 - └── DeprecationWarning
 - └── EncodingWarning
 - └── FutureWarning
 - └── ImportWarning
 - └── PendingDeprecationWarning
 - └── ResourceWarning
 - └── RuntimeWarning
 - └── SyntaxWarning
 - └── UnicodeWarning
 - └── UserWarning\

Podemo criar nossas próprias exceções mas ... falaremos sobre isto na nossa próxima aula, após vermos POO em **Python**.

Às vezes, em uma instrução `try ... except`, pode ser importante trabalhar com a própria mensagem gerada pela exceção.

Isso pode ser feito com a palavra-chave `as`. Veja o mesmo exemplo:

```
In [45]: try:
        print("4/2 = ", divNum(4, 2))
        print("4/0 = ", divNum(4, 0))
        print("4/'0' = ", divNum(4, "0"))
    except TypeError as e:
        print("O tipo de erro é: ", type(e))
        print("A mensagem de erro: ", e)
```

```
4/2 =  2.0
4/0 =  1e+100
O tipo de erro é:  <class 'TypeError'>
A mensagem de erro:  0 divisor não pode ser uma string
```

A sintaxes completa do `try` é:

```
In [46]: #geraErro = False
        geraErro = True
        try:
            print("Tentar executar isto primeiro ...")
            if geraErro:
                novaVar = 1/0
        except:
            print("Executar isto apenas se aconteceu um erro !!!")
        else:
            print("Executar isto apenas se não aconteceu um erro !!!")
        finally:
            print("Executar isto sempre !!!")
```

```
Tentar executar isto primeiro ...
Executar isto apenas se aconteceu um erro !!!
Executar isto sempre !!!
```

Iteradores

Já apresentamos a instrução `for` utilizada para implementar estruturas de repetição com ajuda de um iterador. Até o momento utilizamos apenas dois iteradores: o `range` e o `enumerate`

```
In [47]: itera = range(10)
         print(type(itera))
         for i in itera:
             print(i, end='')
```

```
<class 'range'>
0123456789
```

Como vimos nos exemplos anteriores, os iteradores são fundamentais para percorrer listas

```
In [48]: L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
         for chr in L:
             print(chr.upper(), end=' ')
```

```
A B C D E F G H
```

Esta sintaxe " `for x in y` " nos permite repetir alguma operação para cada valor da lista. O fato de a sintaxe do código ser tão próxima de sua descrição em inglês ("*para [cada] valor na lista*") é apenas uma das escolhas sintáticas que torna o Python uma linguagem tão intuitiva de aprender e usar.

Mas o comportamento aparente não é de fato o que *realmente* está acontecendo. Quando você escreve algo como " `for val in L` ", o interpretador Python verifica se ele possui uma interface *iterator*, que você mesmo pode verificar com a função `iter` :

```
In [49]: itera = iter(L)
         print(type(itera))
         print(itera)
         for chr in itera:
             print(chr.upper(), end=' ')
```

```
<class 'list_iterator'>
<list_iterator object at 0x7fc96010d990>
A B C D E F G H
```

O iterador fornece a funcionalidade exigida pelo loop `for` para funcionar. O objeto `iter` é um contêiner que lhe dá acesso ao próximo objeto enquanto ele for válido, o que pode ser reproduzido utilizando a função `next` :

```
In [50]: itera = iter(L)
         print(next(itera))
         print(next(itera))
         print(next(itera))
         print(next(itera))
```

```
a
b
c
d
```

`range` , da mesma forma que as listas gera um iterator:

```
In [51]: itera = iter(range(10))
         print(next(itera))
         print(next(itera))
         print(next(itera))
         print(next(itera))
```

```
0
1
2
3
```

Uma das vantagens desta abordagem baseada no uso de iteradores é que *a lista completa nunca é*

criada explicitamente!

Podemos ver isso fazendo um cálculo num intervalo que sobrecarregaria a memória do nosso sistema se realmente alocássemos a lista em questão.

```
In [52]: N = 10 ** 100
         for i in range(N):
             if i >= 10:
                 break
             print(i, end=' ')
         print()
```

0 1 2 3 4 5 6 7 8 9

Outros iteradores importantes

Já vimos o `enumerate` em exemplos com listas:

```
In [53]: for i in range(len(L)):
         print(i, L[i])

         # melhor com enumerate
         print("_____")
         for i, vale in enumerate(L):
             print(i, vale)
```

0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h

0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h

Quando se trabalha com múltiplas listas relacionadas, pode ser necessário iterar simultaneamente nelas. Nestes casos podemos utilizar o iterador `zip`, que compacta os iteráveis:

```
In [54]: nomes = ['Ana', 'Beatriz', 'Carlos', 'Daniel', 'Eduardo', 'Fernanda', 'Gabriel', 'Hug']
         for char, name in zip(L, nomes):
             print(char, '-', name)
```

a - Ana
b - Beatriz
c - Carlos
d - Daniel
e - Eduardo
f - Fernanda
g - Gabriel
h - Hugo

Podemos também mapear uma função em cada um dos elementos de uma lista com `map`

```
In [55]: """
         def invert(x):
```

```

        return 1/x;
    """
    invert = lambda x: 1/x

    LInt = list(range(1, 11))
    print(LInt)
    for val in map(invert, LInt):
        print("{:1.3f}".format(val), end=' ')

    print()

```

```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
1.000 0.500 0.333 0.250 0.200 0.167 0.143 0.125 0.111 0.100

```

Também podemos filtrar os elementos de uma lista com uma função que retorne `True` ou `False`

```

In [56]: def ePar(x):
        return x % 2 == 0;

        for val in filter(ePar, LInt):
            print(val, end=' ')

```

```

2 4 6 8 10

```

Já vimos que quando colocamos um asterisco (`*`) na frente de um parâmetro de uma função, estamos indicando que se trata de uma lista. Com os iteradores podemos fazer o mesmo e passar eles como argumentos de funções.

```

In [57]: itera = map(invert, LInt)
        # veja a diferença entre
        print(itera)
        # e
        print(*itera)

```

```

<map object at 0x7fc96010d720>
1.0 0.5 0.3333333333333333 0.25 0.2 0.16666666666666666 0.14285714285714285 0.125 0.11
1111111111111111 0.1

```

O módulo `itertools` contém uma série de iteradores úteis; vale a pena explorar o módulo para ver o que está disponível. Como exemplo, considere a função `itertools.permutations`, que itera sobre todas as permutações de uma sequência:

```

In [58]: from itertools import permutations
        p = permutations(range(3))
        print(*p)

```

```

(0, 1, 2) (0, 2, 1) (1, 0, 2) (1, 2, 0) (2, 0, 1) (2, 1, 0)

```

Da mesma forma, o `itertools.combinations` itera sobre todas as combinações únicas de `N` valores dentro de uma lista:

```

In [59]: from itertools import combinations
        c = combinations(range(4), 2)
        print(*c)

```

```

(0, 1) (0, 2) (0, 3) (1, 2) (1, 3) (2, 3)

```

```

In [60]: import itertools as it
        dir(it)

```

```
Out [60]: ['__doc__',
           '__loader__',
           '__name__',
           '__package__',
           '__spec__',
           '_grouper',
           '_tee',
           '_tee_dataobject',
           'accumulate',
           'chain',
           'combinations',
           'combinations_with_replacement',
           'compress',
           'count',
           'cycle',
           'dropwhile',
           'filterfalse',
           'groupby',
           'islice',
           'pairwise',
           'permutations',
           'product',
           'repeat',
           'starmap',
           'takewhile',
           'tee',
           'zip_longest']
```

List Comprehensions

Procurando por exemplos e tirando dúvidas na Internet você pode ter se deparado com um tipo de construção bem interessante em **Python**: *list comprehensions*. Veja um exemplo:

```
In [61]: pares = [i for i in range(20) if i % 2 == 0]
print(pares)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

As list comprehensions são uma maneira de compactar uma estrutura de repetição `for`, utilizada na construção de lista, em uma única linha curta e legível.

Veja por exemplo como gerar uma lista com os caracteres de uma *string*:

```
In [62]: nome = "Esbel Valero Orellana"
lNome = []
for char in nome:
    lNome.append(char)
print(lNome)

# Utilizando list comprehensions
lNome = [char for char in nome]
print(lNome)
```

```
['E', 's', 'b', 'e', 'l', ' ', 'V', 'a', 'l', 'e', 'r', 'o', ' ', 'O', 'r', 'e', 'l',
'l', 'a', 'n', 'a']
['E', 's', 'b', 'e', 'l', ' ', 'V', 'a', 'l', 'e', 'r', 'o', ' ', 'O', 'r', 'e', 'l',
'l', 'a', 'n', 'a']
```

Vejam que a leitura deste list comprehensions, como outras construções em **Python**, é muito simples: construa uma lista com todos os caracteres da string.

As list comprehensions trabalham com estruturas `for` que por sua vez estão baseadas no uso de iteradores. As list comprehensions pode ser utilizadas para iterar operar com múltiplos iteradores:

```
In [63]: conv = [[a, b] for a in [0,1] for b in [0,1]]
print(conv)
```

```
[[0, 0], [0, 1], [1, 0], [1, 1]]
```

O mecanismo de geração da lista também pode incluir uma condição.

```
In [64]: pares = [i for i in range(20) if i % 2 == 0]
```

Vejam uma diferença fundamental entre as implementações deste problema

```
In [65]: %timeit [i for i in range(1000000) if i % 2 == 0]
```

```
def pares(N):
    pares = []
    for i in range(N):
        if i % 2 == 0:
            pares.append(i)
```

```
%timeit pares(1000000)
```

50.2 ms ± 380 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

60.8 ms ± 468 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

O mecanismo de list comprehensions pode ser utilizado para gerar outros tipos de dados. Basta mudar os delimitadores:

```
In [66]: # Gerando conjuntos
conj = {2**i for i in range(20)}
print(conj)
```

```
{128, 1, 2, 256, 4, 512, 1024, 2048, 8, 4096, 32768, 131072, 262144, 524288, 8192, 16, 16384, 32, 64, 65536}
```

```
In [67]: # Gerando dicionários
digits = {d:ord(d) for d in '0123456789'}
print(digits)
```

```
{'0': 48, '1': 49, '2': 50, '3': 51, '4': 52, '5': 53, '6': 54, '7': 55, '8': 56, '9': 57}
```