Aula 3 - Conceitos básicos Python II - ED e Funções

3.1 Estruturas de dados imbutidas

As estruturas de dados em python são simples porém eficazes. As estruturas de dados mais usadas são **Tuplas**,**lists**, **dict**

3.1.1Tuplas

Uma Tupla é uma sequência **imutável**, de tamanho fixo, de objetos Python. A forma mais fácil para se criar uma tupla é com uma sequencia separada por virgulas.

In [1]:
```
tupla = 1,2,4,8,"ok"
print(tupla)
```

```
(1, 2, 4, 8, 'ok')
```

Geralmente usamos parênteses para criar tuplas mais complicadas, como tuplas de tuplas.

In [2]:
```
tupla_aninhada = (10,20,30),"esta tupla tem uma tupla, uma estring e uma lista", [1,2,3]

print("Número de elementos da tupla : ", len(tupla_aninhada))
print("Número de elementos da tupla dentro da tupla : ", len(tupla_aninhada[0]))
print("Primeiro elemento da tupla : ", tupla_aninhada[0])
print("Segundo elemento da tupla : ", tupla_aninhada[1])
print("Terceiro elemento da tupla : ", tupla_aninhada[2])
```

```
Número de elementos da tupla :  3
Número de elementos da tupla dentro da tupla :  3
Primeiro elemento da tupla :  (10, 20, 30)
Segundo elemento da tupla :  esta tupla tem uma tupla, uma estring e uma lista
Terceiro elemento da tupla :  [1, 2, 3]
```

Podemos converteer qualquer elemento em uma tupla chamando o metodo **tuple()**.

In [3]:
```
lista = ["Serei","Convertida","Em","Uma","Tupla"]
print("Esta é uma lista : ",lista)

tupla_de_lista = tuple(lista)
print("Agora é uma tupla : ", tupla_de_lista)
```

```
Esta é uma lista :  ['Serei', 'Convertida', 'Em', 'Uma', 'Tupla']
Agora é uma tupla :  ('Serei', 'Convertida', 'Em', 'Uma', 'Tupla')
```

Como a tupla é imutável, não podemos substituir os elementos dela:

In [4]:
```
tupla_aninhada = (10,20,30),"esta tupla tem uma tupla, uma estring e uma lista", [1,2,3]
tupla_aninhada[0] = "Alternado o primeiro elemento"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [4], in <cell line: 2>()
      1 tupla_aninhada = (10,20,30),"esta tupla tem uma tupla, uma estring e uma lista", [1,2,3]
----> 2 tupla_aninhada[0] = "Alternado o primeiro elemento"

TypeError: 'tuple' object does not support item assignment
```

Porém, se o elemento em si for mutável, podemos altera-lo *in place*. Por exemplo uma lista, podemos inserir ou remover elementos de uma lista dentro de uma tupla, porém não podemos subtituir a lista por outro Objeto:

In [ ]:
```
print("Tupla antes de inserir um elemento na lista",tupla_aninhada)

tupla_aninhada[2].append(4)
print("Tupla após inserir um elemento na lista",tupla_aninhada)
```

Desempacotando tuplas

Uma das vantagens de se usar tuplas é o seu desempacotamento, ou seja, a atribuição dos elementos das tuplas em variáveis.

In [ ]:
```
tupla_desempacotada = (10,20,30,40,50)
a, b, c, d, e = tupla_desempacotada
print("Os elementos da tupla foram atribuidos às variaveis :", a,b,c,d,e)
```

Ainda, podemos necessitar somente dos valores iniciais da tupla, de forma que todo o resto deve ser desempacotado em uma variável. Para isso podemos usamos um nome de variável com um asterisco*. Todo o resto que não for atribuido às variáveis será

In [ ]:
```
tupla_desempacotada = (10,20,30,40,50)
a, b, *resto = tupla_desempacotada
print("O resto possui então : ", resto)
```

Iterando pelos elementos

Podemos iterar os elementos de uma lista usando a sintaxe **in** ou ainda pelos proprios indices:

In [ ]:
```
tupla1 = (0,1,2,3,4,5)
for e in tupla1:
    print(e)

for i in range(0, len(tupla1)):
    print(tupla1[i])
```

3.1.2 Listas

Em oposição às tuplas, as listas têm tamanhos variáveis e seu conteúdo pode ser modificado in-place. Podemos defini-las usando colchetes [ ] ou com a função **list()**.

In [ ]:
```
lista1 = [1,2,3,5,4]
lista2 = list((1,2,3,5,4))
```

Podemos inserir elementos à uma lista de duas formas distintas:

- append(): insere o novo elemento no fim da lista
- insert(): insere um novo elemento em uma determinada posição. O indice de inserção deve estar entre 0 e o tamanho da lista

In [ ]:
```
lista1.append(5)
print(lista1)

lista2.insert(3, "novo elemento na posição 3")
print(lista2)
```

Para remover elementos podemos uar dois métodos:

- remove( ): Remove um elemento pelo seu valor
- pop( ): remove um elemento com base em seu índice

In [ ]:
```
lista1 = [1,2,3,5,4]
print("Lista antes da remoção : ", lista1)

lista1.remove(3)
print("Lista após a remoção da elemento '3' : ", lista1)

lista1.pop(0)
print("Lista após a remoção do elemento de indice 0 : ", lista1)
```

Usando remove, podemos tentar remover um elemento que não existe, o que origina um erro:

In [ ]:
```
lista1 = [1,2,3,5,4]
lista1.remove("elemento não existe")
```

Para tratar isso, podemos primeiro verificar se o elemento existe na lista, e se existit executamos a remoção do mesmo:

In [ ]:
```
lista1 = [1,2,3,5,4]

# Verificamos se o elemento está na lista
if "elemento não existe" in lista1:
    lista1.remove("elemento não existe")
    print("Elemento removido com sucesso!")
else:
    print("Elemento não está contido na lista")
```

Operações com listas

Como em tuplas, somar duas listas com + as concatena:

In [ ]:
```
lista1 = [1,2,3,5,4]
lista2 = [True, False, (1,2,3)]

lista_concatenada = lista1 + lista2

print("A soma das listas é :", lista_concatenada)
```

A multiplicação por um escalar *n* simplesmente repete a lista *n* vezes.

In [ ]:
```
lista1 = [1,2,3]
print(lista1*2)
```

Ordenação

Uma lista pode ser ordenada in-place (sem criar um novo objeto), com o método **sort()**.

In [ ]:
```
lista1 = [1,2,3,5,4,-1,8,100,5,0.5,-5,0.55]
print("Lista sem ordenação : ", lista1)

lista1.sort()
print("Lista ordenada : ", lista1)
```

Podemos ordenar de forma decrescente adicionando um argumento à função:

In [ ]:
```
lista1 = [1,2,3,5,4,-1,8,100,5,0.5,-5,0.55]
print("Lista sem ordenação : ", lista1)

lista1.sort(reverse =  True)
print("Lista ordenada : ", lista1)
```

Fatiamento

Podemos selecionar seções da maioria dos tipos de sequência usando a notação de fatias (slices), que em seu formato básico é constituida de start:stop passado ao operador [ ]. Os elementos do fatiamento NÃO INCLUEM o indice stop:

In [ ]:
```
lista1 = [0,1,2,3,4,5]
print(lista1[0:2]) # O elemento de indice 2 NÃO foi impresso
```

Se start não é especificado o valor default é 0, se stop não é definido o valor default é o último elemento da lista:

In [ ]:
```
lista1 = [0,1,2,3,4,5]
print(lista1[:2])
print(lista1[3:])
```

Iterando pelos elementos

Podemos iterar os elementos de uma lista usando a sintaxe **in** ou ainda pelos proprios indices:

In [ ]:
```
lista1 = [0,1,2,3,4,5]
for e in lista1:
    print(e)

for i in range(0, len(lista1)):
    print(lista1[i])
```

3.1.3 Dict

dict provavelmente é a estrutura de dados embutida mais importante de Python. Um nome mais comum para ela é *hash map*. Um dict (dicionário) consiste de pares *chave-valor* de tamanho flexível, em que *chave* e *valor* são objetos Python. Uma abordagem para se criar um dicionário é usar chaves { } e dois pontos (:) para separar chaves e valores. Diferentes pares chave:valor são separados por virgula (,). Dicionários são estruturas **mutáveis**.

In [ ]:
```
dic1 = {"chave1":2, "chave2":3, "chave3":4, "ch4":[1,2,3,4]}
print(dic1)
```

Note que os as chaves e valores não precisam ser do mesmo tipo. Para inserir um novo par chave:valor é utilizada a mesma sintaxe para acessar os elementos de uma lista.

In [ ]:
```
dic1["Nova chave"] = "Novo valor"
print(dic1)
```

Podemos ainda criar dicionários com a palavra reservada **dict()**:

In [ ]:
```
novo_dic = dict()
```

Uma das grandes vantagens dos dicionários é que podemos recuperar os valores pelas suas chaves, com a mesma notação das listas e tuplas, porém no lugar do índice usamos a chave:

In [ ]:
```
print(dic1["chave1"])
print(dic1["chave2"])
```

Ainda, podemos acessar as propriedades dos elementos aninhados dentro das chaves. Por exemplo, inserindo um novo elemento na lista armazenada com a chave "lista":

In [ ]:
```
dic1 = {"chave1":2, "chave2":3, "chave3":4, "lista":[1,2,3,4]}
dic1["lista"].append(10)
print(dic1)
```

Se tertarmos acessar uma chave que não existe, um erro será emitido:

In [ ]:
```
dic1["chave_inexistente"]
```

Assim, podemos primeiro verificar se a chave existe e, em caso positivo, realizamos a operação no valor.

In [ ]:
```
dic1 = {"chave1":2, "chave2":3, "chave3":4, "lista":[1,2,3,4]}
if "chave_inexistente" in dic1:
    print(dic1["chave_inexistente"])
else:
    print("Chave não existe")
```

Podemos apagar uma chave usando a palavra **del**.

In [ ]:
```
dic1 = {"chave1":2, "chave2":3, "chave3":4, "lista":[1,2,3,4]}
print("Antes da remoção : ", dic1)
del dic1["chave1"]
print("Após a remoção da chave1 : ", dic1)
```

Os métodos **keys()** e **values()** oferecem iteradores para as chaves e para os valores do dicionário, respectivamente. Embora os pares chave:valor não estejam em uma ordem particular, essas funções devolvem as chaves e os valores na mesma ordem.

In [ ]:
```
dic1 = {"chave1":2, "chave2":3, "chave3":4, "lista":[1,2,3,4]}

for chave in dic1.keys():
    print(chave)

for valor in dic1.values():
    print(valor)
```

In [ ]:
```
dic1 = {"chave1":2, "chave2":3, "chave3":4, "lista":[1,2,3,4]}

# Podemos ainda capturar os iteradores transformando-os em listas:
lista_de_chaves = list(dic1.keys())
print(lista_de_chaves)
print(lista_de_valores)
```

Criando dicionários a partir de sequências

É comum ocasionalmente acabar com duas sequências cujos elementos você queira parear em um dicionário. Para isso usamos a função **zip** em um dicionário:

In [ ]:
```
lista_de_chaves = ["chave1","chave2", "chave3"]
lista_de_valores = [2,3,4]

novo_dicionario = dict(zip(lista_de_chaves,lista_de_valores))
print(novo_dicionario)
```

Tipos de chaves válidas para dicionários

Enquanto os valores de um dicionário podem ser qualquer objeto Python, as chaves devem ser int, float, string ou tuplas. Se uma tupla for usada como chave, os objetos da tupla devem ser imutáveis.

3.1.4 List e dict comprehension (abrangência)

List comprehension

List comprehension (abrangências de lista) são um dos recursos mais amados da linguagem Python. Elas permitem que você componha uma nova lista de modo conciso, filtrando os elementos de uma coleção, transformando os elementos ao passar pelo filtro, com uma expressão concisa. As list comprehension assumem o seguinte formato básico:

```
result = [expr for val in collection if condition]
```

Isso é equivalente ao laço a seguir:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

Por exemplo, dad uma lista de strings, poderiamos filtrar, eliminado aquelas de tamanho 2 ou menores, e também converter as strings para letras maiúsculas assim:

In [ ]:
```
strings  = ['a', 'as','bat','car','dove','python']
nova_lista = [x.upper() for x in strings if len(x) > 2]
print(nova_lista)
```

Ainda, podemos omitir a condição. Considere a forma usual de se criar uma lista com os inteiros de 0 a 10:

In [ ]:
```
lista_inteiros = []
for i in range(11):
    lista_inteiros.append(i)

print(lista_inteiros)
```

Podemos escrever a mesma lista com as comprehensions da seguinte forma:

In [ ]:
```
lista_inteiros = [i for i in range(11)]
print(lista_inteiros)
```

Dict comprehension

Assim como em listas, podemos criar dict comprehensions, com o seguinte aspecto:

```
dict_comp = {expr-chave:expr-valor for value in collection if condition}
```

Da mesma forma, a condição pode ser omitida. Considere a criação de um dicionário em que as chaves são números inteiros de 0 a 10 e os valores são os valores das chaves ao quadrado. Da forma usual o dicionário poderia ser criado da seguinte forma:

In [ ]:
```
dic_potencia = dict()
for i in range(11):
    dic_potencia[i] = i**2

print(dic_potencia)
```

Com as comprehensions podemos criar o mesmo dicionário da seguinte forma:

In [ ]:
```
dic_potencia2 = {i:i**2 for i in range(11)}
print(dic_potencia2)
```

List comprehensions aninhadas

Considere uma lista de listas de inteiros (podemos pensar nela como uma matriz). Queremos armazenar em uma nova lista somente os números negativos das listas. Para isso podemos usar (de forma normal) dois laços for:

In [ ]:
```
matriz = [[1,2,4,7,8],
          [-1,-10,10,-8,8],
          [-1,-10,10,-8,8]]

lista_valores_negativos = []
for i in range(len(matriz)):
    for j in range(len(matriz[i])):
        if(matriz[i][j] < 0):
            lista_valores_negativos.append(matriz[i][j])

print("O valores negativos da lista são: ", lista_valores_negativos)
```

Podemos reduzir um laço e usar uma comprehension. Fazemos a iteração para cada linha, e com esta linha, ao invés de criar um novo laço para verificar elemento a elemento (como criado acima), usamos uma comprehension e um filtro para elementos menores do que zero. Como a comprehension está dentro do laço, se não salvarmos os seus valores são perdidos, de forma que criamos uma lista externa, e a cada nova linha verificada salvamos o resultado nesta lista externa.

In [ ]:
```
matriz = [[1,2,4,7,8],
          [-1,-10,10,-8,8],
          [-1,-10,10,-8,8]]

lista_valores_negativos = []
for linhas in matriz:
    v_negativos_linha = [v for v in linhas if v < 0]  # esta lista é apagada no fim do laço
    lista_valores_negativos.extend(v_negativos_linha) # por isto extendemos a lista 'lista_valores_negativos'

print(lista_valores_negativos)
```

Podemos usar ainda mais uma comprehension para encapsular o laço externo...o resultado final fica assim:

In [ ]:
```
matriz = [[1,2,4,7,8],
          [-1,-10,10,-8,8],
          [-1,-10,10,-8,8]]

lista_valores_negativos = [valor for linhas in matriz for valor in linhas if valor < 0]
print(lista_valores_negativos)
```

Pode parecer complicado aninhar comprehensions (e é mesmo!), mas podemos pensar o seguinte: primeiro colocar o valor que queremos avaliar, em seguida colocamos todos os laços, do mais externo ao mais interno. Isso facilita muito a organização do código. No exemplo acima, da forma usual o código demandou 5 linhas (2 **for** aninhados), já com as comprehensions somente uma!

Exercícios I

1. Crie uma lista com os elementos `[10,20,30,1,-1,50,800,500,-600,800,65,78,52]`, transforme a lista em uma tupla e imprima o número de elementos e o maior elemento da mesma.
2. Considerando a tupla criada no exemplo anterior. Você deseja armazenar os 4 primeiros valores em 4 variáveis (a,b,c,d), e o resto não te serve para nada. Crie um código que implemente este desempacotamento de variáveis.
3. Escreva um código que leia 5 valores digitados pelo usuário e salve-os em uma tupla chamada T. Imprima o menor valor digitado.
4. Crie uma lista com a tupla T gerada no exercicio anterior, porém com os elementos na ordem inversa.
5. Crie as seguintes listas: `A = [[1,2,3],[4,5,6],[7,8,9]]  B = [[-1,2,0],[8,5,6],[10,8,9]]`
   A. Calcule A + B.
   B. Calcule A - B.
   C. Calcule 3 * A.
   D. Calcule a soma da diagonal principal de A e de B e realize a soma das duas.
   E. Troque a linha 0 pela linha 2 da matriz A.
   F. Multiplique a linha 2 da Matriz B por 3.
6. Crie uma lista com os primeiros 70 números da sequência de Fibonacci. Os primeiros número são `1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377`.
7. Calcule a soma dos 70 primeiros números da sequência de Fibonacci.
8. Crie uma lista (usando **for** e **list comprehension**) contendo somente os números pares da sequência de Fibonacci com 70 elementos.
9. Considere as seguintes sequências matemáticas, e para cada uma delas escreva um algoritmo que armazene os elementos em uma lista usando um laço **for** e a **list comprehension**
   A. $\{n\}, n=1,\ldots,100$
   B. $\left\{\frac{n}{n+1}\right\}, n=1,\ldots 100 = \left\{\frac{1}{2},\frac{2}{3},\frac{3}{4},\ldots\right\}$
   C. $\left\{\frac{(-1)^n(n+1)}{3^n}\right\}, n=1,\ldots 100 = \left\{-\frac{2}{3},\frac{3}{9},-\frac{4}{27},\ldots\right\}$
10. Crie um dicionário em que as chaves são os elementos da série da letra A do ex. anterior, e os valores a série da letra B.
11. Considere um banco de dados relativo as vendas de peças em uma fábrica. A Tabela abaixo mostra o tipo de peça e as vendas:

Peça	Vendas
AR001	50
AR002	50
CR003	60
BR001	85

   A. Coloque todas as informações em um dicionário.
   B. Crie um código que peça ao usuário o nome de uma peça e mostre uma mensagem referente ao número de vendas da peça.
   C. Se o usuário digitar um código de peça que não existe, uma mensagem deve alertar o usuário que a peça não existe.
   D. Ainda, se a peça não existir, o programa deve perguntar ao usuário se ele gostaria de criar um novo cadastro; se a resposta for "não" o programa é finalizado, caso contrário ele deve pedir ao usuário para digitar o código e a quantidade vendida, inserir os dados na estrutura, e imprimir todos os códigos com as vendas.
12. Considere uma linha de produção que registra os problemas ocorridos em uma codificação numérica, cada número se refere a um erro. Os seguintes valores ocorreram em uma semana: `[1,2,1,1,4,5,3,6,5,5,8,7,4,5,2,2,2,9,6,3,5,8,7,4,7,4,7,4,4,7,4,1,4,5,2]`. Crie um algoritmo que calcule a frequência de ocorrência de cada erro, bem como a probabilidade de ocorrência do erro (a frequência/total de erros). Uma mensagem deve ser exibida avisando o usuário do erro que mais ocorre, e a sua probabilidade de ocorrência em uma semana.
13. Para cada uma das estruturas de dados abaixo, diga qual o seu tipo e acesse os elementos:
   A. `ed1 = ["A","E","I","O","U"]`
   B. `ed2 = [("A","E"),("I","O","U"),("F",1,2)]`
   C. `ed3 = [[1,2,3,4], [2,3,2,1], [2,3,2,1,1]]`
   D. `ed3 = {"A":ed1, "B":ed2 , "C":ed3}`
   E. `ed4 = ([1,2,3],["a,b,s"], {"C1":[2,22,2], "C2":"C3"})`
   F. `ed5 = [ed3, ed4]`
14. Ainda considerando as ED, o que será impresso com os seguintes acessos:
   A. `ed1 = ed5[0]["B"][1]`
   B. `ed2 = ed4[2]["C2"]`

3.2 Funções

Funções são trechos de código reutilizáveis. Se você precisa executar uma parte do código diversas vezes, ele é um bom condidato a ser escrito como uma função. A sintaxe para a função exige a palavra reservada `def`, seguida do nome da função e parênteses. Dentro dos parênteses coloca-se os argumentos e logo após dois pponntos (lembre que em Python tudo que vem após dois pontos é um bloco, e portanto deve ser *indentado*. O código abaixo cria uma função que soma dois números.

In [ ]:
```
def soma(a, b):
    resultado = a + b
    print("A soma é : {}".format(resultado))
```

A função só é executada quando a mesma é *chamada*:

In [ ]:
```
soma(2,3)
```

3.2.1 Retornando valores

Como em todas as linguagens de programação, as funções em Python podem ou não retornar valores. Isso é realizado com a palavra *return*. O código abaixo realiza a soma de dois valores e retorna o resultado ou invés de imprimi-lo.

In [ ]:
```
def soma2(a,b):
    resultado = a + b
    return resultado

# usando a função
resul = soma2(int(input("Digite o primeiro valor : ")),int(input("Digite o segundo valor : ")))
print("A soma é {}".format(resul))
```

Em todas as linguagens de programação, as funções podem retornar somente um valor. Como a estrutura de dados *tuplas* é muito usada, e também muito prática, é muito comum as funções em Python retornarem diversos valores como uma tupla, o que parece (erroneamente) que a função retorna múltiplos valores. A função abaixo recebe 2 números e calcula a soma, subtração, multiplicação e divisão de um pelo outro, retornando tudo como uma tupla com 4 valores.

In [ ]:
```
def calculadora(a,b):
    soma = a + b
    sub  = a - b
    mult =  a * b
    div =  a/b
    return soma, sub, mult, div

sm,sb,mt,dv = calculadora(2.0, 3.0)
print(sm, sb, mt, dv)
```

3.2.2 Escopo de funções

As variáveis declaradas nas funções "morrem" assim que a função termina. Ou seja, uma variável declarada em uma função não pode ser usada fora dela, e vice versa. O exemplo abaixo declara uma variável fora da função e tenta alterar o seu valor dentro dela.

In [ ]:
```
a = 10

def usando_a():
    a = 30

usando_a()
print(a)
```

Para conseguirmos alterar a variável a dentro da função, a mesma deve ter um escopo *global*. Para isso usamos a palavra `global` na variável dentro da função que está definida fora dela.

In [ ]:
```
a = 10

def usando_a():
    global a
    a = 30

usando_a()
print(a)
```

3.2.3 Funções como objetos

As funções (como tudo em Python) são objetos, o que quer dizer que podemos atribuir funções à variáveis. Considere o código abaixo, que atribui a sunção `calculadora(a, b)` a uma variável chamada cálculo:

In [ ]:
```
# Atribuindo uma função a uma variável
calculos = calculadora
calculos(10,20)
```

Dessa forma, podemos então atribuir funções a listas, criando uma lista de funções. O código abaixo atribui todas as funções que criamos até agora em uma lista:

In [ ]:
```
lista_de_funcoes = []
lista_de_funcoes = [soma, soma2, calculadora, usando_a]

# Usamos a função em uma determinada posição, passando os argumentos

#Na posicao 0 está a 'soma'
lista_de_funcoes[0](10,20)

#Na posicao 0 está a 'soma'
lista_de_funcoes[0](10,20)
```

Ainda, como as funções são objetos, podemos usar as próprias funções como argumentos de outras funções. Considere o caso em que temos uma função que calcula integrais numericamente. Sabemos que as integrais são calculadas com base em uma função específica, porém o método numérico é o mesmo. Dessa forma podemos criar uma função que calcula as integrais e recebe um argumento referente à função que terá sua integral calculada. A função abaixo calcula a integral da função $f(x) = x^2$ pelo método dos trapézios.

In [ ]:
```
def calcula_integral_x2(a,b,n):
    # Calcula a integral da função x^2, de 'a' a 'b' com n intervalos
    passo =  (b - a)/n
    x = a
    v_y = []
    while x < b + passo:
        x = x + passo
        v_y.append(x**2)

    integral = 0
    for i in range(0, len(v_y)-1):
        integral =  integral + (v_y[i + 1] + v_y[i])*passo/2
    return integral

a = 1.0
b = 3.0
integral = (3**3)/3  -  (1**3)/3

print("A integral real é : ", integral)
print("A integral aproximada com 10 retangulos é : ", calcula_integral_x2(a,b,10))
print("A integral aproximada com 10 retangulos é : ", calcula_integral_x2(a,b,100))
print("A integral aproximada com 10 retangulos é : ", calcula_integral_x2(a,b,1000))
print("A integral aproximada com 10 retangulos é : ", calcula_integral_x2(a,b,10000))
print("A integral aproximada com 10 retangulos é : ", calcula_integral_x2(a,b,100000))
```

Usando a função acima, só podemos calcular a integral de $f(x) = x^2$. Neste caso, podemos alterar o código para receber uma função como argumento, e esta será usada para calcular os valores do vetor v_y.

OBS:

A passagem de argumentos em funções de Python não funciona exatamente como outras linguagens, que possuem as possibilidades de passagem por referência e por valor. É considerada uma boa prática em Python retornar valores e fazer a reatribuição, caso seja necessário alterar algum argumento.

In [ ]:
```
def calcula_integral_generica(a,b,f,n):
    # Calcula a integral da função x^2, de 'a' a 'b' com n intervalos
    passo =  (b - a)/n
    x = a
    v_y = []
    while x < b + passo:
        x = x + passo
        v_y.append(f(x))

    integral = 0
    for i in range(0, len(v_y)-1):
        integral =  integral + (v_y[i + 1] + v_y[i])*passo/2
    return integral

#definimos a função x^2
def x2(x):
    return x**2

#definimos a função x^2
def f(x):
    return x**2 + 1/x

print("A integral aproximada de x2 com 10000 retangulos é : ", calcula_integral_generica(a,b,x2,10000))
print("A integral aproximada de f com 10000 retangulos é : ", calcula_integral_generica(a,b,f,10000))
```

Exercícios II

1. Crie uma lista com os elementos `[10,20,30,1,-1,50,800,500,-600,800,65,78,52]`. Para cada uma das tarefas abaixo, crie uma função que receba uma lista e imprima o resultado (e aplique as funções à lista):
```
1.1. Realizar a soma dos elementos
1.2. Encontrar o maior elemento
1.3. Encontrar o menor elemento
1.4. Calcular a média
```
1. Considerando as funções criadas acima, salve todas em uma lista chamada `lista_func`, e usando um laço `for` aplique todas as funções na lista de elementos acima.
2. Crie as seguintes tarefas como funções, que recebem uma lista de listas (matriz) numérica como argumento, e retornam a matriz com os valores alterados:
```
3.1 multiplica_linha(l, p): multiplica a linha *l* pelo número *p*.
3.2 troca_linhas(l1,l2): troca as linhas *l1* por *l2*.
3.3 soma_com_linha(l1,l2): altera os valores da linha 2, somando l1 (l2 = l2 + l1).
```
Use a matriz abaixo para realizar os testes nas funções criadas: `A = [[1,2,3],[4,5,6],[7,8,9]]`
3. Escreva uma função que calcule o fatorial de n (`fat(n)`).
4. Crie funções (uma separada da outra) para as seguintes operações (as funções devem retornar o resultado) da operação:
```
5.1 soma(a,b): soma a e b.
5.2 subtracao(a,b): subtrai b de a.
5.3 multiplicacao(a,b): multiplica a por b.
5.4 divisao(a,b): divide a por b.
5.5 potenciacao(a,b): eleva a a potência de b
5.6 fat(n): calcule o fatorial de n.
```
5. Crie um aplicativo que funcione como uma calculadora para o usuário: o programa mostra um 'menu' com as seguintes alternativas:
```
a. soma(a,b): soma a e b.
b. subtracao(a,b): subtrai b de a.
c. multiplicacao(a, b): multiplica a por b.
d. divisao(a,b): divide a por b.
e. potenciacao(a,b): eleva a a potência de b
f. fat(n): calcule o fatorial de n.
```
E o usuário deve digitar uma letra. Dependendo da letra que o usuário digitar, uma das funções criadas é chamada (note que para cada função os argumentos devem também ser coletados pelos programa)
6. Incremente a calculadora criada no exercício anterior, oferecendo ao usuário a possibilidade de refazer as contas ou sair do programa ao final de um cálculo. DICA: Use um laço `while` para controlar se o programa deve parar ou não, e mais um input do usuário ao final dos cálculos que defina se ele vai sair ou não.