


Módulo de Programação Python: Introdução à Linguagem

Segundo Encontro

Apresentação de Python: Estruturas de controle de fluxo

 No description has been provided for this image

Objetivo: Explorar o uso de tipos de dados avançados em python: dicionários e conjuntos. Explorar a implementação de estruturas de controle de fluxo, estruturas condicionais e de repetição, em python.

Tipos de dados estruturados nativos

Além dos tipos mais simples **Python** fornece um conjunto de tipos de dados estruturados muito rico e interessante.

Nome	Exemplo	Descrição
list	[1, 2, 3]	Coleção ordenada
tuple	(1, 2, 3)	Coleção imutável ordenada
dict	{'a':1, 'b':2, 'c':3}	Mapeamento do tipo (chave, valor)
set	{1, 2, 3}	Coleção não ordenada de valores únicos

Vamos continuar a nos debruçar sobre cada um destes tipos.

Na aula anterior apresentamos listas e tuplas.

```
In [1]: L_int = [1, 2, 3, 4]           # Lista de inteiros
        L_str = ['Nome', 'EMail', 'CPF'] # Lista de strings
        L_mix = [1, 'Nome', 3.14]      # Lista mista
        L_list_int = [[1, 2, 3], [4, 5, 6]] # Lista de listas de inteiros
        l_list = [L_int, L_str, L_mix, L_list_int] # Lista de listas

        print('L_int =', L_int)
        print('L_str =', L_str)
        print('L_mix =', L_mix)
        print('L_list_int =', L_list_int)
        print('l_list =', l_list)
```

```
L_int = [1, 2, 3, 4]
L_str = ['Nome', 'EMail', 'CPF']
L_mix = [1, 'Nome', 3.14]
L_list_int = [[1, 2, 3], [4, 5, 6]]
l_list = [[1, 2, 3, 4], ['Nome', 'EMail', 'CPF'], [1, 'Nome', 3.14], [[1, 2, 3], [4, 5, 6]]]
```

```
In [2]: point2D = (1.0, 1.0)
        point3D = (1.0, 1.0, 1.0)
        registro = ('Nome', 'EMail', 'CPF')
        tupla_tuplas = (point2D, point3D, registro)
        print('point2D =', point2D)
        print('point3D =', point3D)
```

```
print('registro =', registro)
print('tupla_tuplas =', tupla_tuplas)
```

```
point2D = (1.0, 1.0)
point3D = (1.0, 1.0, 1.0)
registro = ('Nome', 'EMail', 'CPF')
tupla_tuplas = ((1.0, 1.0), (1.0, 1.0, 1.0), ('Nome', 'EMail', 'CPF'))
```

```
In [3]: lista_tuplas = [point2D, point3D, registro]
        tupla_listas = (L_int, L_str, L_mix, L_list_int)
        print('lista_tuplas =', lista_tuplas)
        print('tupla_listas =', tupla_listas)
```

```
lista_tuplas = [(1.0, 1.0), (1.0, 1.0, 1.0), ('Nome', 'EMail', 'CPF')]
tupla_listas = ([1, 2, 3, 4], ['Nome', 'EMail', 'CPF'], [1, 'Nome', 3.14], [[1, 2, 3], [4, 5, 6]])
```

```
In [4]: #litas são mutáveis
        print('L_int =', L_int)
        L_int[0] = 'Nome'
        print('L_int =', L_int)
```

```
L_int = [1, 2, 3, 4]
L_int = ['Nome', 2, 3, 4]
```

```
In [5]: #tuplas são imutáveis
        print('point2D =', point2D)
        try:
            point2D[0] = 2.0
        except TypeError as err:
            print('TypeError:', err)
        print('point2D =', point2D)
```

```
point2D = (1.0, 1.0)
TypeError: 'tuple' object does not support item assignment
point2D = (1.0, 1.0)
```

Dicionários

Este é, provavelmente, o tipo mais interessante entre os tipos de dados estruturados nativos de **Python**.

O dicionário não parece com quase nada do que encontramos em outras linguagens mais tradicionais. Para declarar um dicionário utilizamos um conjunto de pares chave valor, no formato `{chave:valor, }`, separados por vírgula e delimitado por chaves.

O acesso aos elementos do dicionário utiliza mais uma vez índices, mas desta vez não se trata de um mecanismo de indexação numérico começando em zero. Os índices dos dicionários são as chaves.

Os dicionários são tipos de dados mutáveis.

Veja os seguintes exemplos:

```
In [6]: # Declarando um dicionário
        meuReg = {'Nome': "Jonas", 'SobreNome': "Oliveira", "Idade":25, "Altura":1.85}
        # Acessando um elemento do dicionário
        print("Nome: ", meuReg['Nome'])
        # Os dicionarios são mutáveis
        # Modificando um elemento do dicionario
        meuReg['Nome'] = "Juninho"
        # Adicionando um novo elemento
        meuReg['Peso'] = 88.3
        # Imprimindo o dicionário
        print("Registro completo: ", meuReg)
```

```
print(meuReg.keys())
print(meuReg.values())
```

Nome: Jonas

Registro completo: {'Nome': 'Juninho', 'SobreNome': 'Oliveira', 'Idade': 25, 'Altura': 1.85, 'Peso': 88.3}

dict_keys(['Nome', 'SobreNome', 'Idade', 'Altura', 'Peso'])

dict_values(['Juninho', 'Oliveira', 25, 1.85, 88.3])

Conjuntos

Finalmente os conjuntos são declarados como as listas, substituindo os colchetes por chaves.

Para os matemáticos pode ficar mais simples entender este tipo de dado se os associamos a conjuntos matemáticos, para os quais estão definidos uma série de operações bem conhecidas.

Os conjuntos, embora mutáveis, não são ordenados. Portanto, não há espaço para indexação. A indexação e o slicing não podem alterar ou acessar um item de um conjunto, pois um conjunto python não oferece suporte para indexação em conjuntos.

Veja os exemplos a seguir:

```
In [7]: # Declarando conjuntos
P = { 2, 4, 5, 6, 8, 10}
#P = { 2, 4, 6, 8, 10}
I = {1, 3, 5, 7, 9}
try:
    print("P[0] = ", P[0])
except TypeError as err:
    print('TypeError:', err)
# União de conjuntos
num = P.union(I) # equivalente a P | I
nulo = I.intersection(P) # equivalente a I & P
difPdeI = P.difference(I) # equivalente a P - I
difIdeP = I.difference(P) # equivalente a I - P
print("Conjunto P, ", P)
print("Conjunto I, ", I)
print("União, ", num)
print("Intersecção, ", nulo)
print("Diferença, ", difPdeI)
print("Diferença, ", difIdeP)
```

TypeError: 'set' object is not subscriptable

Conjunto P, {2, 4, 5, 6, 8, 10}

Conjunto I, {1, 3, 5, 7, 9}

União, {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Intersecção, {5}

Diferença, {2, 4, 6, 8, 10}

Diferença, {1, 3, 9, 7}

Estruturas de controle de fluxo

Com o conteúdo apresentado até aqui somos capazes de entender um pouco o **Python** e brincar, de forma bastante limitada, com um interpretador como o **IPython**.

Para andar soltos no mundo, implementando algoritmos, se faz necessário entender como controlar uma sequência de comandos de forma a executar determinados blocos de acordo com condições específicas, ou então de forma repetitiva, recorrente ou recursiva.

Nos próximos tópicos abordaremos de forma rápida estruturas condicionais, como **if**, **elif** e **else**, e estruturas de repetição, incluindo os tradicionais **for** e **while**.

Estrutura condicional: if - elif - else:

Estruturas condicionadas, geralmente conhecidas como comandos `if-then`, permitem que o código execute um determinado bloco de instruções apenas se uma determinada condição for satisfeita.

Em **Python** se introduz a tradicional estrutura `if-else`, acrescentando apenas o `elif` que é uma contração para quando seja necessário utilizar um `else if` (o clássico `if` aninhado).

Veja alguns exemplos simples:

```
In [8]: # podemos utilizar uma variável bool diretamente
condição = True
#condição = False
# veja como usar uma condição if simples
x = 0.0
print("x = ", x)

#if condição:
if condição:
    x = 3.14
# Após a estrutura condicional
print("x = ", x)

# Podemos utilizar também uma condição com duas alternativas
if x > 0:
    x = -1
else:
    x = 1
# Após a nova estrutura condicional
print("x = ", x)

# Ou estruturas condicionais aninhadas
if x == 0:
    print(x, "zero")
elif x > 0:
    print(x, "positivo")
elif x < 0:
    print(x, "negativo")
else:
    print(x, "não é um valor numérico...")
```

```
x = 0.0
x = 3.14
x = -1
-1 negativo
```

```
In [9]: print("Voce quer ir na festa?")
idade = int(input("Qual a sua idade? "))

if idade > 70:
    print("Você está muito velho para festa!!! Vai tomar seu remedio e dormir.")
elif idade < 1:
    print("Nem saiu das fraldas e está querendo ir para a farra!!!")
elif idade >= 18:
    print("Bem vindo na festa!")
else:
    "Ta muito novinho. Vai dormir que é melhor."
```

```
Voce quer ir na festa?
Bem vindo na festa!
```

Da versão **3.10** em diante, o **Python** implementou um recurso semelhante ao `switch-case` chamado "correspondência de padrões estruturais". Você pode implementar esse recurso com as palavras-chave `match-case`.

Algumas pessoas debatem se `match` e `case` são ou não palavras-chave em Python. Isso ocorre porque você pode usar ambos como nomes de variáveis e funções. Mas isso é outra história para outro dia.

Você pode referir-se a ambas as palavras-chave como "palavras-chave suaves", se desejar.

Vejam como se fazia antes do `match-case`

```
In [10]: diaSemana = 1

if diaSemana == 1:
    print("Segunda-feira")
elif diaSemana == 2:
    print("Terça-feira")
elif diaSemana == 3:
    print("Quarta-feira")
elif diaSemana == 4:
    print("Quinta-feira")
elif diaSemana == 5:
    print("Sexta-feira")
elif diaSemana == 6:
    print("Sábado")
elif diaSemana == 7:
    print("Domingo")
else:
    print("Dia inválido")
```

Segunda-feira

Para escrever instruções switch você pode usar a sintaxe abaixo:

```
match term:
    case pattern-1:
        action-1
    case pattern-2:
        action-2
    case pattern-3:
        action-3
    case _:
        action-default
```

Veja que o símbolo de sublinhado é utilizado para definir um caso padrão.

```
In [11]: diaSemana = 1

match diaSemana:
    case 1:
        print("Segunda-feira")
    case 2:
        print("Terça-feira")
    case 3:
        print("Quarta-feira")
    case 4:
        print("Quinta-feira")
    case 5:
        print("Sexta-feira")
    case 6:
        print("Sábado")
    case 7:
        print("Domingo")
    case _:
        print("Dia inválido")
```

Estrutura de repetição for

Estruturas de repetição ou laços, são utilizados para executar um determinado bloco de instruções repetidas vezes. O tradicional construtor de estruturas de repetição `for` está disponível com uma abordagem um pouco diferentes. Veja um exemplo simples em **Python**:

```
In [12]: PpI = [1, 3, 5, 7, 9]
print("PpI: [ ", end='')
for x in PpI:
    print(x, end=' ') # imprimindo todos os elementos na mesma linha
print(""])
```

PpI: [1 3 5 7 9]

Reparem que neste exemplo utilizamos o operador `in` para ligar a variável de controle do laço `x`, com o conjunto de valores possíveis (pode se ler "para cada valor x que pertencem à lista PpI).

Veja um outro exemplo:

```
In [13]: listaCompras = ['ovos', 'farinha', 'leite', 'maças']
for item in listaCompras:
    print(item)
```

ovos
farinha
leite
maças

Seria interessante podermos enumerar os itens da lista, sobre tudo no contexto de uma lista de compras. A função `enumerate()` pode ajudar. Esta função gera um iterador, a partir da lista, que retorna uma dupla contendo o cada elemento precedido de seu índice.

Veja p exemplo:

```
In [14]: for index, item in enumerate(listaCompras):
        print(index, item)
```

0 ovos
1 farinha
2 leite
3 maçãs

O mesmo código poderia ser implementado utilizando a função `range()`.

A função `range()` retorna um *iterator* de números inteiros.

```
In [15]: itera = range(10)
print(type(itera))
print(itera)
```

```
<class 'range'>
range(0, 10)
```

Como vimos no exemplo anterior, o iterador precisa um parâmetro de entrada que representa o fim do processo iterativo. o iterador gera uma sequência de números inteiros que começa em 0 e vai até o valor final, que não está incluído. Para visualizar os valores gerados podemos transformar o iterador numa lista:

```
In [16]: # a função range retorna um iterator de números inteiros
# pode ser usada definindo apenas o valor final, que não estará incluído
```

```

fim = 10
itera = range(fim)
print("range( ", fim, " ) - ", type(itera), ": ", list(itera))

```

range(10) - <class 'range'> : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

O valor inicial do iterador pode ser modificado. Com isto será necessário passar dois parâmetros para a função.

```

In [17]: # pode ser definindo também o valor inicial, que estara incluido
ini = 2
itera = range(ini, fim)
print("range( ", ini, ", ", fim, " ) - ", type(itera), ": ", list(itera))

```

range(2 , 10) - <class 'range'> : [2, 3, 4, 5, 6, 7, 8, 9]

Repare que o valor inicial está incluído na sequencia gerada.

Pose-se definir ainda um valor para o incremento que, por padrão, é um.

```

In [18]: # pode ainda ser definindo também o valor do incremento
passo = 2
itera = range(ini, fim, passo)
print("range( ", ini, ", ", fim, ", ", passo, " ) - ", type(itera), ": ", list(itera))

```

range(2 , 10 , 2) - <class 'range'> : [2, 4, 6, 8]

Veja como fica o exemplo da lista anterior utilizando `range()`

```

In [19]: # No caso da lista de inteiros
print("PpI: [ ", end=' ')
for i in range(len(PpI)):
    print(PpI[i], end=' ')
print("]")

print()

# No caso da lista de compras
for i in range(len(listaCompras)):
    print(i, listaCompras[i])

```

PpI: [1 3 5 7 9]

0 ovos
1 farinha
2 leite
3 maçãs

O modelo de repetição for é muito útil para iterar sobre sequências, como listas, tuplas, conjuntos, strings, etc.

Veja os exemplo abaixo:

```

In [20]: Nome = "Esbel Tomas Valero Orellana"
for chr in Nome:
    print(chr, end=' ')

print()

for i in range(len(Nome)):
    print(Nome[i], end=' ')

```

E s b e l T o m a s V a l e r o O r e l l a n a
E s b e l T o m a s V a l e r o O r e l l a n a

```
In [21]: tupla = (1, 2, 3, 4, 5)
         for i in tupla:
             print(i, end=' ')

         print()

         for i in range(len(tupla)):
             print(tupla[i], end=' ')
```

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

```
In [22]: P = { 2, 4, 5, 6, 8, 10}
         for i in P:
             print(i, end=' ')
```

```
2 4 5 6 8 10
```

Como fazer para percorrer dicionarios?

```
In [23]: dic = {'Nome': "Jonas", 'SobreNome': "Oliveira", "Idade":25, "Altura":1.85}
         print("Por padrão percoremos as chaves do dicionário")
         for i in dic:
             print(i, end=' ')

         print()
         print("Podemos indicar que queremos percorrer as chaves de forma explícita")

         for i in dic.keys():
             print(i, end=' ')

         print()
         print("Podemos indicar que queremos percorrer os valores de forma explícita")

         for i in dic.values():
             print(i, end=' ')

         print()
         print("Podemos indicar que queremos percorrer os itens de forma explícita")

         for i in dic.items():
             print(i, end=' ')
```

Por padrão percoremos as chaves do dicionário

Nome SobreNome Idade Altura

Podemos indicar que queremos percorrer as chaves de forma explícita

Nome SobreNome Idade Altura

Podemos indicar que queremos percorrer os valores de forma explícita

Jonas Oliveira 25 1.85

Podemos indicar que queremos percorrer os itens de forma explícita

('Nome', 'Jonas') ('SobreNome', 'Oliveira') ('Idade', 25) ('Altura', 1.85)

Estrutura de repetição while

A estrutura `while`, como em **C/C++**, repete um bloco de código enquanto uma determinada condição for satisfeita.

Veja o exemplo:

```
In [24]: #Estrutura de repetição controlada por contador
         i = 0 #inicializando o contador
         print("PpI: [ ", end='')
         while i < len(PpI): # verificandop a condição de parada
             print(PpI[i], end=' ')
             i += 1 #incrementando o contador
         print(""])
```



```
#Estrutura de repetição controlada por sentinela
i = 0 #inicializando o contador
print("PpI: [ ", end='')
while True:
    try:
        print(PpI[i], end=' ')
    except:
        break
    i += 1 #incrementando o contador
print(""])
```

PpI: [1 3 5 7 9]

PpI: [1 3 5 7 9]

Como vimos no exemplo anterior as estruturas de repetição podem ter sua execução modificada pelo uso de duas instruções: `break` e `continue`. Veja como modificar o código do laço `for` apresentado na seção anterior:

```
In [25]: print("PpI: [", end='')
for i in range(len(PpI)):
    if i%2 == 0 :
        continue #pula para o fim da iteração atual e vai para a próxima iteração
    print(PpI[i], end=' ')
print(""])
```

PpI: [3 7]

Uma outra característica que pode ser comentada sobre as estruturas de repetição em **Python** é a possibilidade de utilizar um estrutura `else` no final, tanto dos laços `for` quanto dos `while`.

A utilização prática ou mesmo a necessidade de se ter este recurso desperta muitas dúvidas entre os programadores. Vamos apenas colocar um exemplo para documentar o uso do mesmo.

```
In [26]: # Uma situação simples baseada no exemplo anterior
print("PpI: [", end=' ')
for i in range(len(PpI)):
    if i > 10:
        break # sai completamente da execução do laço
    print(PpI[i], end=' ')
else:
    print(""])
```

PpI: [1 3 5 7 9]