Introdução a Python

Iágson Carlos Lima Silva
iagsoncarlos@lapisco.ifce.edu.br

O que é o Python?

O Python é uma linguagem interpretada, o que reduz o ciclo de edição/teste/depuração porque não há necessidade de uma etapa de compilação. Para executar aplicativos de Python, você precisa de um ambiente de runtime/interpretador para executar o código.

Tipos de dados

Tipos de dados

Numéricos:

int:

Representa números inteiros, como 1, -5, 100, etc.

float:

Representa números de ponto flutuante, incluindo números decimais, como 3.14, -0.5, etc.

complex:

Representa números complexos na forma "real + imaginário", como 2 + 3j.

Tipos de dados

Sequenciais:

str:

Representa sequências de caracteres, como "Olá, mundo!".

list:

Representa uma lista ordenada e mutável de valores, por exemplo, [1, 2, 3].

tuple:

Semelhante a uma lista, mas é imutável, ou seja, seus elementos não podem ser alterados após a criação, por exemplo, (1, 2, 3).

Tipos de dados

Mapeamentos:

dict:

Representa um dicionário, que é uma estrutura de chave-valor, por exemplo, {'nome': 'Jonh', 'idade': 33}.

Tipos de dados

Booleanos:

bool:

Representa valores booleanos, ou seja, True (verdadeiro) ou False (falso).

Tipos de dados

Conjuntos:

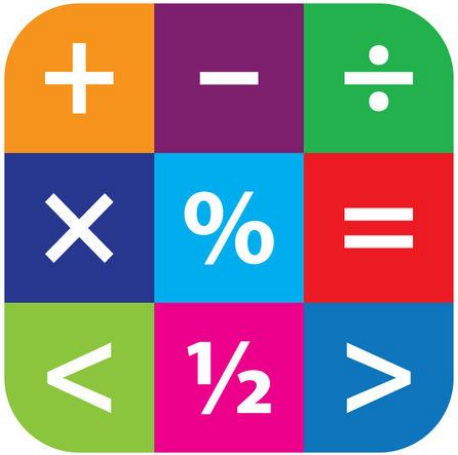
set:

Representa um conjunto não ordenado de elementos únicos, por exemplo, {1, 2, 3}.

frozenset:

Similar a um conjunto, mas é imutável.

Operadores



Operadores Aritméticos

Operador	Nome	Função
+	Adição	Realiza a soma de ambos operandos.
-	Subtração	Realiza a subtração de ambos operandos.
*	Multiplicação	Realiza a multiplicação de ambos operandos.
/	Divisão	Realiza a Divisão de ambos operandos.
//	Divisão inteira	Realiza a divisão entre operandos e retorna um inteiro.
%	Módulo	Retorna o resto da divisão de ambos operandos.
**	Exponenciação	Retorna o resultado da elevação da potência pelo outro.

Operadores de Comparação

Operador	Nome	Função
==	Igual a	Verifica se um valor é igual ao outro
!=	Diferente de	Verifica se um valor é diferente ao outro
>	Maior que	Verifica se um valor é maior que outro
>=	Maior ou igual	Verifica se um valor é maior ou igual ao outro
<	Menor que	Verifica se um valor é menor que outro
<=	Menor ou igual	Verifica se um valor é menor ou igual ao outro

Operadores de Atribuição

Operador	Equivalente a
=	x = 1
+=	x = x + 1
-=	x = x - 1
*=	x = x * 1
/=	x = x / 1
%=	x = x % 1

Operadores Lógicos

Operador	Definição
and	Retorna True se ambas as afirmações forem verdadeiras
or	Retorna True se uma das afirmações for verdadeira
not	Retorna Falso se o resultado for verdadeiro

Operadores de Identidade

Operador	Definição
is	Retorna True se ambas as variáveis são o mesmo objeto
is not	Retorna True se ambas as variáveis não forem o mesmo objeto

- Exemplo:

```
lista = [1, 2, 3]
outra_lista = [1, 2, 3]
recebe_lista = lista

print(f"São o mesmo objeto? {lista is recebe_lista}") # True
print(f"São o mesmo objeto? {lista is outra_lista}") # False
```

Operadores de Associação

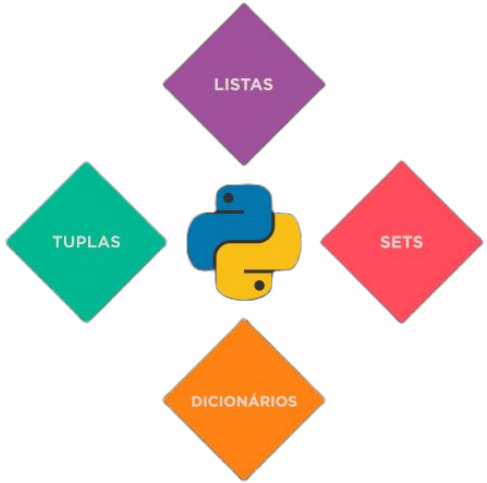
Operador	Definição
in	Retorna True caso o valor seja encontrado na sequência
in not	Retorna True caso o valor não seja encontrado na sequência

- Exemplo:

```
lista = ["Python", 'Academy', "Operadores", 'Condições']

print('Python' in lista)  # True
print('SQL' not in lista) # True
```

Introdução a Estruturas de Dados



Tuplas

Tuplas é uma sequência ordenada e imutável de elementos.

- Criamos tuplas usando parênteses: `tupla = (item1, item2)`.
- Elementos acessados por índices, assim como listas.
- Ideal para dados que não mudam, como coordenadas.

- Exemplo:

```
coordenadas = (10, 20)
x = coordenadas[0]  # Retorna 10
```

Listas

Lista é uma sequência ordenada de elementos.

- Criamos listas usando colchetes: lista = [item1, item2, item3].
- Os elementos são acessados por índices (começando em 0).

- Exemplo:

```
frutas = ["maçã", "banana", "laranja"]
primeira_fruta = frutas[0]  # Retorna "maçã"
```

Listas

Manipulação de Listas:

- **Adicionar:** `lista.append(item)` para adicionar um item ao final.
- **Inserir:** `lista.insert(indice, item)` para inserir um item em um índice específico.
- **Remover:** `lista.pop(indice)` para remover e retornar um item por índice.
- **Modificar:** `lista[indice] = novo_item` para substituir um item.
- **Tamanho:** `len(lista)` retorna o número de itens na lista.

Compreensão de Listas

- Uma maneira concisa de criar ou transformar listas.
- Substitui loops tradicionais por uma única linha.
- Mais legível e eficiente para tarefas simples.
- Sintaxe: `[retorno iterável]`

- Exemplo:

```
numeros = [1, 2, 3, 4, 5]
quadrados = [x ** 2 for x in numeros]
```

Compreensão de Listas

- Usa uma expressão e um loop for.
- Pode incluir condicionais para filtrar elementos.
- Sintaxe: `[retorno iterável condição]`

- Exemplo:

```
numeros = [1, 2, 3, 4, 5]
pares = [x for x in numeros if x % 2 == 0]
```

Dicionário

Dicionários são coleções não ordenadas de pares chave-valor.

- Criamos dicionários usando chaves: dicionario = {"chave1": valor1, "chave2": valor2}.
- Acessamos valores através das chaves, não dos índices.
- Ótimo para representar associações.

- Exemplo:

```
aluno = {"nome": "Maria", "idade": 23}
nome_do_aluno = aluno["nome"]  # Retorna "Maria"
```

Dicionário

Manipulação de Dicionários:

- **Adicionar:** `dicionario[nova_chave] = novo_valor` para adicionar um novo par.
- **Atualizar:** `dicionario[chave_existente] = novo_valor` para modificar um valor existente.
- **Remover:** `del dicionario[chave_a_remover]` para excluir um par chave-valor.
- **Chaves:** `dicionario.keys()` retorna uma lista de chaves.
- **Valores:** `dicionario.values()` retorna uma lista de valores.

Sets

Sets são coleções não ordenadas de elementos únicos.

- Criamos sets usando chaves: `conjunto = {elemento1, elemento2, elemento3}`.
- Úteis para manter elementos únicos e realizar operações de conjuntos.

- Exemplo:

```
numeros = {1, 2, 3, 4, 5}
```

Sets

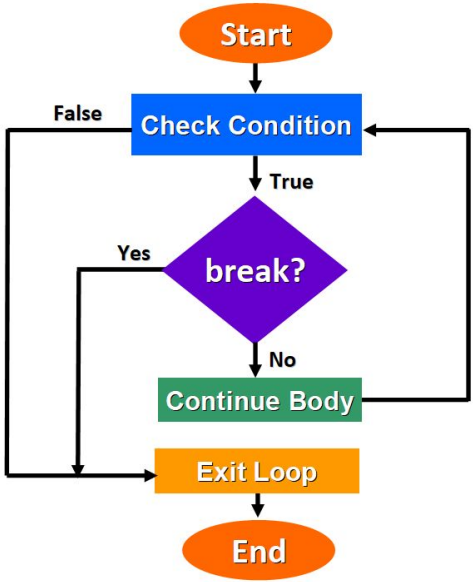
Manipulação de Sets:

- Adicionar: `conjunto.add(elemento)` para adicionar um elemento.
- Remover: `conjunto.remove(elemento)` para remover um elemento.
- União de conjuntos: `conjunto1 | conjunto2` para obter a união de dois conjuntos.
- Interseção de conjuntos: `conjunto1 & conjunto2` para obter a interseção.
- Diferença de conjuntos: `conjunto1 - conjunto2` para obter a diferença.

Qual estrutura escolher?

- **Listas:** Para coleções mutáveis e ordenadas.
  - Exemplo: Armazenar itens em um carrinho de compras.
- **Tuplas:** Para coleções imutáveis e ordenadas.
  - Exemplo: Representar coordenadas geográficas.
- **Dicionários:** Para associações chave-valor.
  - Exemplo: Mapear informações de um usuário.
- **Sets:** Para coleções de elementos únicos e operações de conjuntos.
  - Exemplo: Remover duplicatas de uma lista e operações de conjuntos.

Estruturas de Decisão e Repetição



Estruturas de Decisão: if, elif, else

Usadas para tomar decisões com base em condições.

- **if** verifica uma condição e executa um bloco de código se verdadeiro.
- **elif** adiciona condições alternativas.
- **else** captura qualquer outra condição.

Exemplo:

```
idade = 18
if idade < 18:
    print("Menor de idade")
elif idade == 18:
    print("Tem 18 anos")
else:
    print("Maior de idade")
```

Estruturas de Repetição: for

- Utilizada para iterar sobre uma sequência (listas, tuplas, strings, etc.).
- Executa um bloco de código para cada item na sequência.

- Exemplo:

```
numeros = [1, 2, 3, 4, 5]
for num in numeros:
    print(num)
```

Estruturas de Repetição: for

Função range():

- Cria uma sequência de números.
- Pode especificar início, fim e passo.
- Útil para loops for e gerar índices.

- Exemplo:

```
# Começa em 1, vai até 9, passo de 2
for i in range(1, 10, 2):
    print(i)
```

Estruturas de Repetição: for

Função `enumerate()`:

- Cria um objeto enumerado que contém pares (índice, valor).
- Útil para obter índices durante iteração.
- "*Start*": `enumerate(iterável, start=1)`, inicia com o índice 1.

- Exemplo:

```
frutas = ["maçã", "banana", "laranja"]
for indice, fruta in enumerate(frutas):
    print(f"No índice {indice}: {fruta}")
```

Estruturas de Repetição: while

- Executa um bloco de código enquanto uma condição for verdadeira.
- Útil quando não sabemos quantas vezes precisamos repetir.

- Exemplo:

```
contador = 0
while contador < 5:
    print(contador)
    contador += 1
```

Controle de Fluxo

- Estruturas de decisão e repetição controlam o fluxo de execução do programa.
- Permitem que o programa se adapte a diferentes situações.

- Exemplo:

```
for num in range(1, 6):
    if num % 2 == 0:
        print(f"{num} é par")
    else:
        print(f"{num} é ímpar")
```

Manipulação de Arquivos

Manipulação de Arquivos

- **"r" (Read):** Modo de leitura.

  Abre o arquivo para leitura. O arquivo deve existir.

- **"w" (Write):** Modo de escrita.

  Abre o arquivo para escrita. Se o arquivo já existir, seu conteúdo será apagado. Se não existir, um novo arquivo será criado.

- **"a" (Append):** Modo de anexação.

  Abre o arquivo para adicionar conteúdo ao final. Se o arquivo não existir, um novo arquivo será criado.

- **"r+" (Read and Write):** Modo de leitura e escrita.

  Abre o arquivo para leitura e escrita. O arquivo deve existir.

Manipulação de Arquivos: Bloco with

Abrindo e Lendo Arquivos:

- Usamos o bloco with com a função `open()` para abrir arquivos.
- Dentro do bloco, podemos ler o conteúdo do arquivo.
- Também podemos escrever e anexar usando o bloco with.

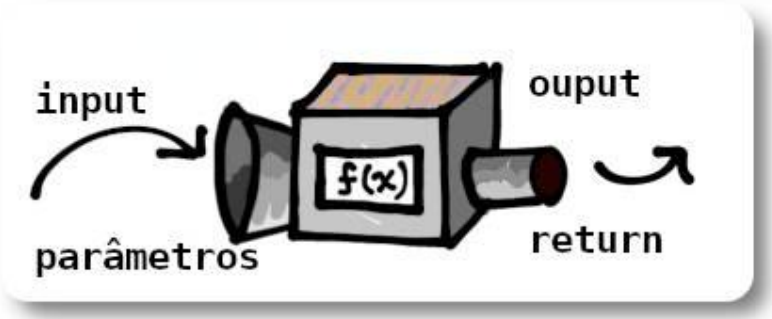
- Exemplo:

```
with open("novo.txt", "w") as novo_arquivo:
    novo_arquivo.write("Olá, mundo!")
with open("anexo.txt", "a") as arquivo_anexo:
    arquivo_anexo.write("Mais conteúdo.")
```

Manipulação de Arquivos: Bloco with

- O bloco `with` é a abordagem recomendada para manipular arquivos.
- Garante o fechamento adequado dos arquivos após a manipulação.
- Melhora a legibilidade e a segurança do código.

Funções



Funções

- Um bloco de código nomeado e reutilizável.
- Recebe argumentos como entrada (opcional).
- Executa um conjunto de instruções.
- Retorna um valor (opcional).

- Exemplo:

```
def minha_funcao(nome):
    return f"Olá, {nome}!"
```

Funções Lambda

- Também chamadas de funções anônimas.
- São pequenas expressões sem nome.
- Úteis para tarefas simples e rápidas.

- Exemplo:

```
soma = lambda a, b: a + b # soma(x, y)
```

Funções Lambda

Usos Comuns de Funções Lambda:

- Ordenação de listas com chave personalizada.
  - Exemplo: `ordenados = sorted(numeros, key=lambda x: x)`
- Transformações simples em listas com `map()`.
  - Exemplo: `quadrado = map(lambda x: x ** 2, numeros)`
- Filtragem de elementos em listas com `filter()`.
  - Exemplo: `pares = filter(lambda x: x % 2 == 0, numeros)`

Gerenciamento de Erros

- Erros que ocorrem durante a execução do programa.
- Interrompem o fluxo normal do programa.
- Python gera mensagens de erro (tracebacks) para ajudar na identificação.

- Exemplo:

```
print(10 / 0)  # Exceção de divisão por zero
```

```
Traceback (most recent call last):
  File "/home/user/Downloads/classes/python/main.py", line 1, in <module>
    print(10 / 0)
ZeroDivisionError: division by zero
```

Gerenciamento de Erros

Blocos *Try-Except*:

- Usados para capturar exceções e evitar falhas do programa.
- O código problemático fica dentro do bloco **`try`**.
- O bloco **`except`** captura e lida com a exceção.

- Exemplo:

```
try:
    resultado = 10 / 0
except ZeroDivisionError:
    print("Erro: Divisão por zero!")
```

Gerenciamento de Erros

Diferentes Exceções:

- Pode-se usar múltiplos blocos **except** para tratar exceções específicas.
- O bloco except com exceção genérica (Exception) deve ser o último.

- Exemplo:

```
try:
    valor = int(input("Digite um número: "))
except ValueError:
    print("Erro: Valor inválido!")
except KeyboardInterrupt:
    print("Operação interrompida pelo
usuário.")
except Exception:
    print("Ocorreu um erro inesperado.")
```

Gerenciamento de Erros

Bloco *Else* e *Finally*:

- **`else`**: Executa quando o bloco try não gera exceções.
- **`finally`**: Sempre executado, independentemente de exceções.

- Exemplo:

```
try:
    arquivo = open("dados.txt", "r")
except FileNotFoundError:
    print("Arquivo não encontrado.")
else:
    conteudo = arquivo.read()
    print("Conteúdo:", conteudo)
finally:
    arquivo.close()
```

Introdução ao Numerical Python

Introdução ao Numerical Python

- O NumPy traz o poder computacional de linguagens como C e Fortran para Python
- NumPy é uma biblioteca Python para manipulação de arrays multidimensionais
- É uma ferramenta poderosa para trabalhar com dados numéricos
- Ideal para computação científica, análise de dados e muito mais
- Todos os elementos em uma matriz NumPy devem ser homogêneos

Introdução ao Numerical Python

Criando Vetores:

- Vetores são arrays unidimensionais.
- Podem ser criados a partir de listas ou com funções NumPy.

- Exemplo:

```
import numpy as np

vetor1 = np.array([1, 2, 3, 4, 5]) # Vetor de 1 a 5
vetor2 = np.arange(0, 10, 2)  # Vetor de 0 a 8, de 2 em 2
```

Introdução ao Numerical Python

Criando Matrizes:

- Matrizes são arrays bidimensionais.
- Podem ser criadas a partir de listas de listas ou com funções NumPy.

- Exemplo:

```
import numpy as np
matriz1 = np.array([[1, 2, 3], [4, 5, 6]])
matriz2 = np.zeros((3, 4))  # Matriz 3x4 preenchida com zeros
matriz2 = np.ones((3, 4))  # Matriz 3x4 preenchida com 1
```

Introdução ao Numerical Python

Operações com Vetores e Matrizes:

- NumPy permite realizar operações de forma eficiente.
- Soma, subtração, multiplicação, divisão e mais.

- Exemplo:

```python
import numpy as np

vetor1 = np.array([1, 2, 3])
vetor2 = np.array([4, 5, 6])


soma = vetor1 + vetor2 # [5 7 9]
subtracao = vetor1 - vetor2 # [-3 -3 -3]
multiplicacao = vetor1 * vetor2 # [ 4 10 18]
divisao = vetor1 / vetor2 # [0.25 0.4  0.5 ]
```

Introdução ao Numerical Python

Funções Estatísticas Básicas:

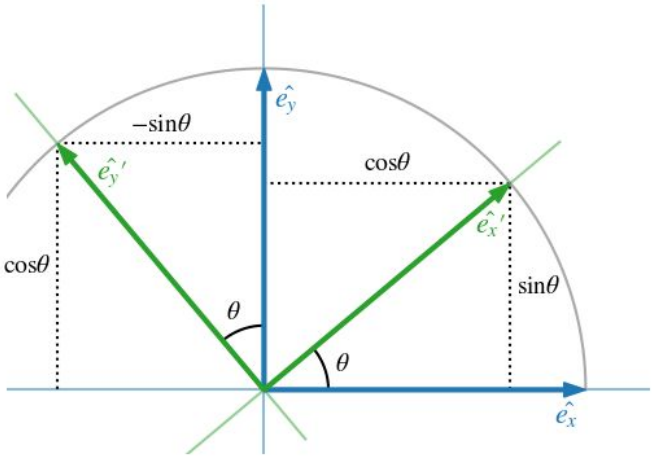
- NumPy oferece funções para cálculos estatísticos simples.
- Média, mediana, desvio padrão, mínimo, máximo, e mais.

- Exemplo:

```python
import numpy as np
vetor = np.array([5, 10, 15, 20, 25])

media = np.mean(vetor) # 15.0
mediana = np.median(vetor) # 15.0
desvio_padrao = np.std(vetor) # 7.07
maximo = np.max(vetor) # 25
minimo = np.min(vetor) # 5
```

Introdução à Álgebra Linear com NumPy



Introdução à Álgebra Linear com NumPy

Produto Escalar:

- O produto escalar (ou produto interno) entre dois vetores é a soma dos produtos de seus elementos correspondentes.
- NumPy permite calcular o produto escalar facilmente.

- Exemplo:

```
import numpy as np
vetor1 = np.array([2, 3, 4])
vetor2 = np.array([5, 6, 7])
produto_escalar = np.dot(vetor1, vetor2)  # 2*5 + 3*6 + 4*7 = 56
```

Introdução à Álgebra Linear com NumPy

Produto Vetorial:

- O produto vetorial entre dois vetores resulta em um terceiro vetor perpendicular a ambos.
- NumPy oferece funções para calcular o produto vetorial.

- Exemplo:

```python
import numpy as np
vetor1 = np.array([1, 2, 3])
vetor2 = np.array([4, 5, 6])
produto_vetorial = np.cross(vetor1, vetor2)  # [-3, 6, -3]
```

Introdução à Álgebra Linear com NumPy

Multiplicação de Matrizes:

- A multiplicação de matrizes é uma operação fundamental na álgebra linear.
- NumPy permite multiplicar matrizes de forma eficiente.

- Exemplo:

```
import numpy as np
matriz1 = np.array([[1, 2], [3, 4]])
matriz2 = np.array([[5, 6], [7, 8]])
produto_matrizes = np.dot(matriz1, matriz2)  # [[19, 22], [43, 50]]
```

Introdução à Álgebra Linear com NumPy

Inversa de Matriz:

- A matriz inversa é uma matriz que, quando multiplicada pela matriz original, resulta na matriz identidade.
- NumPy permite calcular a inversa de uma matriz.

- Exemplo:

```
import numpy as np

matriz = np.array([[2, 1], [5, 3]])
matriz_inversa = np.linalg.inv(matriz) # [[ 3. -1.] [-5.  2.]]
```

Referências

Python Software Foundation. Tutorial Python 3. Disponível em: https://docs.python.org/3/tutorial/index.html. Acesso em: (17/08/2023).

NumPy Documentation. Disponível em: https://numpy.org/doc/stable/user/index.html#user. Acesso em: (17/08/2023).

Obrigado pela atenção!

Dúvidas?

Iágson Carlos Lima Silva
iagsoncarlos@lapisco.ifce.edu.br