Python :: Notas de Aula

Prof. Dr. Gilderlanio Santana de Araújo
gilderlanio@gmail.com

2 de setembro de 2024

Conteúdo

1 Conceitos de Linguagens de Programação **5**

1.1 Tipos de linguagens 6

1.1.1 Linguagens de alto nível 6

1.1.2 Linguagens de Baixo Nível 7

1.1.3 Comparação Geral 7

1.2 Conceitos de linguagens de programação 8

1.2.1 Sintaxe e Semântica 8

1.2.2 Paradigmas de Programação 8

1.2.3 Tipos de Dados 9

1.2.4 Variáveis e Escopo 9

1.2.5 Controle de Fluxo 9

1.2.6 Funções e procedimentos 10

1.2.7 Modularidade e abstração 10

1.2.8 Compilação e interpretação 10

1.2.9 Gerenciamento de memória 11

1.2.10 Concorrência e paralelismo 11

1.2.11 Exceções e tratamento de erros 11

1.2.12 Linguagens fortemente e fracamente tipadas 11

2 Ambientes de desenvolvimento **13**

2.1 Tutorial de uso do PyCharm e do Jupyter 13

2.2 PyCharm 13

2.2.1 Instalação 13
2.2.2 Criando um projeto 13
2.2.3 Configurando um ambiente virtual 14
2.2.4 Editor de código 14
2.2.5 Executando código 14
2.2.6 Debug 14
2.3 Project Jupyter 15
2.3.1 Instalação 15
2.3.2 Editor de código 15
2.3.3 Células de código 15
2.3.4 Markdown 15
2.3.5 Executando células 16
2.3.6 Considerações finais 16

3 Variáveis, atribuição e tipagem dinâmica 17
3.1 Regras e convenções 18

4 Números inteiros e flutuantes 19
4.1 Inteiros 19
4.1.1 Função int() 20
4.2 Números flutuantes ou decimais (float) 20
4.3 Operações com números 21
4.3.1 Módulo math 22

5 Strings 25
5.1 Criando *strings* 25
5.2 Indexação 25
5.3 Métodos 26

CAPÍTULO 1

Conceitos de Linguagens de Programação

As linguagens de programação são definidas por diversos conceitos que determinam seu funcionamento, estrutura e como elas facilitam a criação de programas. Esses conceitos incluem a sintaxe, que especifica as regras de escrita dos comandos, e a semântica, que define o significado desses comandos. Além disso, as linguagens de programação podem ser classificadas em diferentes paradigmas, como a programação imperativa, orientada a objetos, funcional e lógica, cada uma com suas próprias abordagens para resolver problemas e construir soluções de software.

Um aspecto importante na escolha de uma linguagem de programação é o nível de abstração que ela oferece. Linguagens de alto nível, como Python e Java, proporcionam maior abstração do hardware, usando sintaxe mais próxima da linguagem humana e facilitando a criação de código legível e fácil de manter. Por outro lado, linguagens de baixo nível, como Assembly, oferecem maior controle sobre os recursos físicos do computador, permitindo otimizações detalhadas para desempenho máximo, mas exigindo um conhecimento mais profundo da arquitetura da máquina e uma sintaxe mais complexa.

Compreender esses conceitos é essencial para determinar a linguagem mais adequada para um projeto específico. A escolha da linguagem depende dos requisitos do projeto, como a necessidade de portabilidade, facilidade de desenvolvimento, controle de hardware e desempenho desejado. Linguagens de alto

nível são geralmente preferidas para aplicações que priorizam rapidez de desenvolvimento e manutenção, enquanto linguagens de baixo nível são escolhidas para aplicações que exigem desempenho crítico e controle preciso sobre os recursos do sistema. Dessa forma, o entendimento claro dos conceitos de linguagens de programação ajuda desenvolvedores a tomar decisões informadas e eficientes na escolha da tecnologia ideal para seus projetos.

1.1 Tipos de linguagens

1.1.1 Linguagens de alto nível

Linguagens de alto nível são aquelas que oferecem uma maior abstração do hardware do computador, tornando o processo de programação mais intuitivo e acessível para os humanos. Elas se aproximam mais da linguagem natural e escondem muitos dos detalhes do funcionamento interno do computador. As principais características incluem:

1. **Abstração:** Proporcionam uma alta abstração dos detalhes do hardware, permitindo que os programadores se concentrem na lógica e funcionalidade do programa em vez de aspectos técnicos específicos da máquina.

2. **Sintaxe simples e intuitiva:** Utilizam sintaxe que é mais próxima da linguagem humana, com palavras-chave e estruturas de controle que facilitam a leitura e escrita do código (e.g., `if`, `for`, `while`).

3. **Portabilidade:** São independentes de plataforma, o que significa que o mesmo código pode ser executado em diferentes tipos de hardware com pouca ou nenhuma modificação. Isso é facilitado pela presença de compiladores ou interpretadores que traduzem o código para a linguagem de máquina específica.

4. **Facilidade de uso:** São projetadas para serem fáceis de aprender e usar, com muitas funcionalidades integradas para manipulação de dados, estruturas complexas e funções matemáticas.

5. **Gestão automática de memória:** Oferecem recursos para gerenciamento automático de memória, como coleta de lixo (*garbage collection*), o que alivia o programador da tarefa de gerenciar manualmente a alocação e liberação de memória.

6. **Exemplos:** Python, Java, C++, C#, Ruby, JavaScript.

1.1.2 Linguagens de Baixo Nível

Linguagens de baixo nível estão mais próximas do hardware e são menos abstratas. Elas oferecem maior controle sobre os recursos físicos do computador, mas requerem um entendimento mais detalhado da arquitetura da máquina. As principais características incluem:

1. **Pouca abstração:** Fornecem comandos que estão diretamente relacionados ao conjunto de instruções do processador, permitindo um controle direto sobre o hardware. Isso significa que o programador deve lidar com detalhes como registradores, endereçamento de memória e instruções de máquina.

2. **Alta eficiência e desempenho:** Como estão muito próximas do hardware, programas escritos em linguagens de baixo nível são geralmente mais rápidos e eficientes, pois podem ser otimizados para usar os recursos da máquina de forma muito específica.

3. **Dependência de plataforma:** São específicas para um tipo de hardware ou arquitetura de processador, o que significa que o código precisa ser adaptado ou completamente reescrito para funcionar em plataformas diferentes.

4. **Sintaxe complexa e difícil de leitura:** O código é muitas vezes difícil de entender e manter, pois utiliza uma sintaxe que não é intuitiva para humanos. Comandos e instruções são frequentemente representados por códigos de operação numéricos ou mnemônicos (abreviações).

5. **Controle manual de memória:** Exigem que o programador gerencie manualmente a alocação e liberação de memória, o que pode levar a erros como vazamentos de memória (*memory leaks*).

6. **Exemplos:** Assembly, linguagem de máquina (código binário).

1.1.3 Comparação Geral

- **Abstração:** Linguagens de alto nível oferecem mais abstração, enquanto as de baixo nível oferecem menos.

- **Facilidade de Uso:** Alto nível é mais fácil para desenvolvimento e manutenção, enquanto baixo nível é mais complexo.
- **Desempenho:** Baixo nível tende a ser mais eficiente em termos de desempenho, mas exige mais esforço do programador.
- **Portabilidade:** Alto nível é geralmente mais portátil, enquanto baixo nível é específico para uma arquitetura.

1.2 Conceitos de linguagens de programação

1.2.1 Sintaxe e Semântica

- **Sintaxe:** Refere-se às regras que definem a estrutura correta de um programa, incluindo como comandos, declarações, expressões e símbolos devem ser organizados. A sintaxe é essencialmente a "gramática"de uma linguagem de programação, especificando como os programas devem ser escritos para serem reconhecidos corretamente pelo compilador ou interpretador.
- **Semântica:** Define o significado das construções sintáticas de uma linguagem. Enquanto a sintaxe descreve a forma, a semântica descreve o comportamento. Por exemplo, a semântica de um comando `if` descreve como o fluxo de controle é alterado com base na avaliação de uma expressão booleana.

1.2.2 Paradigmas de Programação

- **Programação Imperativa:** Baseia-se em comandos que alteram o estado do programa através de instruções sequenciais. Exemplos incluem C e Python. Esse paradigma foca em como o computador deve realizar as tarefas passo a passo.
- **Programação Orientada a Objetos (POO):** Estrutura programas em objetos que combinam dados (atributos) e comportamentos (métodos). Linguagens como Java e C++ são exemplos. A POO promove reutilização de código, encapsulamento, herança e polimorfismo.
- **Programação Funcional:** Centrada na avaliação de funções matemáticas, evita estados mutáveis e efeitos colaterais. Linguagens como Haskell e Lisp

são exemplos de linguagens funcionais. Esse paradigma facilita a paralelização e promove código mais previsível.

- **Programação Lógica:** Baseia-se em regras e lógica, permitindo que problemas sejam resolvidos mediante inferências lógicas. Prolog é um exemplo de linguagem de programação lógica, onde se especificam fatos e regras, e o sistema deduz as respostas.

1.2.3 Tipos de Dados

- Tipos de dados definem os valores que podem ser manipulados em um programa, como inteiros, flutuantes, strings e listas. Eles são fundamentais para a segurança e a precisão do código.

- **Tipagem Estática:** Os tipos de variáveis são determinados em tempo de compilação. Isso ajuda a identificar erros antes da execução do programa. Exemplos incluem C e Java.

- **Tipagem Dinâmica:** Os tipos são definidos em tempo de execução, o que permite maior flexibilidade, mas pode resultar em erros de tempo de execução. Exemplos incluem Python e JavaScript.

1.2.4 Variáveis e Escopo

- **Variáveis:** São usadas para armazenar dados que podem ser usados e manipulados ao longo da execução do programa. As variáveis têm nomes e tipos associados a elas.

- **Escopo:** Refere-se ao contexto dentro do qual uma variável ou função está definida e acessível. Escopos comuns incluem:

  - **Global:** Variáveis acessíveis em qualquer parte do programa.
  - **Local:** Variáveis acessíveis apenas em uma função ou bloco específico.
  - **Em bloco:** Variáveis definidas dentro de blocos de código como loops ou condicionais.

1.2.5 Controle de Fluxo

- Estruturas de controle de fluxo determinam a ordem de execução das instruções em um programa. Principais estruturas incluem:

  - **Condicionais:** Comandos como `if`, `else` e `switch` que permitem tomar decisões com base em condições.
  - **Laços de repetição:** Estruturas como `for`, `repeat`, `while` e `do-while` permitem repetir blocos de código enquanto uma condição for verdadeira.
  - **desvios:** Comandos como `break`, `continue` e `return` que alteram o fluxo de execução de loops ou funções.

1.2.6 Funções e procedimentos

- **Funções:** Blocos de código que recebem entradas (parâmetros), realizam cálculos ou operações e retornam um valor. Funções são fundamentais para a reutilização de código e para a modularização de programas.
- **Procedimentos:** Semelhantes às funções, mas não retornam valores. São usados para executar uma série de instruções sem a necessidade de produzir um resultado.

1.2.7 Modularidade e abstração

- **Modularidade:** É a prática de dividir um programa em partes menores (módulos) que podem ser desenvolvidas, testadas e mantidas independentemente. Isso facilita a manutenção e a escalabilidade do código.
- **Abstração:** Consiste em ocultar detalhes complexos e mostrar apenas aspectos essenciais. Em POO, isso é alcançado através de interfaces e classes, permitindo que os programadores se concentrem no que o objeto faz, e não em como ele faz.

1.2.8 Compilação e interpretação

- **Compilação:** Processo de traduzir o código fonte de um programa para um código de máquina executável pelo computador antes da execução. Exemplos incluem linguagens como C e C++.
- **Interpretação:** Executa o código fonte linha a linha, traduzindo e executando diretamente em tempo de execução. Isso oferece maior flexibilidade e facilidade de depuração, mas pode ser menos eficiente. Exemplos incluem Python e JavaScript.

1.2.9 Gerenciamento de memória

- **Alocação Estática:** A memória é alocada em tempo de compilação e tem um tamanho fixo durante a execução do programa.
- **Alocação Dinâmica:** A memória é alocada em tempo de execução, o que permite que programas usem exatamente o quanto precisam. Pode ser gerenciada manualmente (como em C, usando `malloc` e `free`) ou automaticamente via *garbage collection* (como em Java e Python).

1.2.10 Concorrência e paralelismo

- **Concorrência:** Refere-se à execução de múltiplas tarefas de forma intercalada, em um único núcleo de processamento ou em múltiplos. É útil para melhorar a responsividade de programas.
- **Paralelismo:** Refere-se à execução simultânea de múltiplas tarefas, aproveitando múltiplos núcleos de processamento ou múltiplas máquinas. É utilizado para aumentar a performance de programas.

1.2.11 Exceções e tratamento de erros

- Sistemas de tratamento de erros permitem que um programa lide com situações inesperadas ou condições de erro de forma controlada. Exceções são capturadas usando blocos como `try`, `catch` e `finally`, permitindo que o programa se recupere de erros sem terminar abruptamente.

1.2.12 Linguagens fortemente e fracamente tipadas

- **Fortemente tipada:** Linguagens que aplicam regras rígidas quanto ao tipo dos dados, não permitindo conversões implícitas entre tipos diferentes. Exemplo: Java.
- **Fracamente tipada:** Permitem maior flexibilidade e conversões implícitas entre tipos, o que pode resultar em comportamento inesperado. Exemplo: JavaScript.

CAPÍTULO 2

Ambientes de desenvolvimento

2.1 Tutorial de uso do PyCharm e do Jupyter

Neste tutorial, segue as instruções básicas sobre o uso do PyCharm e o Project Jupyter Lab. Ambas as ferramentas são poderosas para desenvolvimento de software e análise de dados em Python e oferecem recursos úteis em diferentes cenários.

2.2 PyCharm

2.2.1 Instalação

- Baixe e instale o PyCharm Community Edition a partir do site oficial (`https://www.jetbrains.com/pycharm/`).
- Siga as instruções do instalador para concluir a instalação.

2.2.2 Criando um projeto

- Abra o PyCharm e clique em *Create New Project* (Criar novo projeto).
- Escolha um diretório para o projeto.

- Defina o interpretador Python desejado.
- Clique em *Create* (Criar) para criar o projeto.

2.2.3 Configurando um ambiente virtual

- Recomenda-se configurar um ambiente virtual para isolar as dependências do projeto.
- No menu, clique em *File* (Arquivo) $->$ *Settings* (Configurações) $->$ *Python Interpreter* (Interpretador Python).
- Clique no ícone de engrenagem e selecione *Add* (Adicionar).
- Escolha *Virtualenv Environment* (Ambiente Virtualenv) e siga as instruções para configurar o ambiente.

2.2.4 Editor de código

- O PyCharm oferece um editor de código e repleto de recursos.
- Crie um novo arquivo Python clicando com o botão direito do mouse na pasta do projeto e selecionando *New* (Novo) $->$ *Python File* (Arquivo Python).
- Comece a escrever seu código no editor.

2.2.5 Executando código

- De forma simples, para executar um arquivo Python no PyCharm, basta clicar com o botão direito do mouse no arquivo e selecionar *Run* (Executar);

- Se estiver familiarizado, pode executar o arquivo em python pelo terminal com o comando *python* $nome_do_arquivo.py$.

2.2.6 Debug

- O PyCharm oferece recursos avançados de depuração.
- Coloque pontos de interrupção no seu código clicando na margem esquerda do editor.

- Clique em *Debug* (Depurar) para executar o código em modo de depuração.

2.3 Project Jupyter

2.3.1 Instalação

- O Jupyter pode ser instalado utilizando o **pip** com o comando `pip install jupyterlab`.
- O Jupyter Notebook geralmente é instalado junto com a distribuição do Anaconda (`https://www.anaconda.com/`) ou o Miniconda (`https://docs.conda.io/en/latest/miniconda.html`).

2.3.2 Editor de código

- Abra o terminal ou prompt de comando e digite o comando `jupyter-lab ou jupyter lab`, que abrirá uma interface web no seu navegador padrão.
- No Jupyter Notebook, clique em *New* (Novo) $->$ *Python 3* para criar um novo notebook Python.
- Uma nova aba será aberta com o notebook vazio.

2.3.3 Células de código

- O Jupyter Notebook utiliza células para escrever e executar o código.
- Cada célula pode conter código Python que pode ser executado independentemente.
- Digite seu código em uma célula e pressione `Shift + Enter` para executá-la.

2.3.4 Markdown

- Além das células de código, você também pode adicionar células de texto formatado usando a sintaxe Markdown.
- Selecione `Markdown` no menu dropdown na barra de ferramentas superior para criar uma célula de texto.

2.3.5 Executando células

- As células de código podem ser executadas individualmente ou em sequência.
- Para executar uma célula, selecione-a e pressione `Shift + Enter`.
- O resultado da célula será exibido logo abaixo dela.

2.3.6 Considerações finais

Espero que este tutorial básico tenha fornecido uma introdução útil ao uso do PyCharm e do Jupyter. Ambas as ferramentas têm suas próprias vantagens e podem ser úteis em diferentes contextos de desenvolvimento e análise de dados em Python.

CAPÍTULO 3

Variáveis, atribuição e tipagem dinâmica

Em Python, as variáveis são usadas para armazenar e representar valores. Uma variável é basicamente um nome associado a um valor específico. Elas são usadas para armazenar, manipular dados e realizar cálculos em um programa. Elas são dinamicamente tipadas, permitindo que você atribua diferentes tipos de valores ao codificar.

Em Python, você pode definir uma variável simplesmente atribuindo um valor a ela usando o operador de atribuição '='. Por exemplo:

```
1 >>> nome = "Gil Araujo"
2 >>> idade = 35
3 >>> altura = 1.75
```

Detalhes sobre os tipos de dados serão dados no próximo capítulo. Por enquanto abstraia! Nesse exemplo, três variáveis são definidas: nome, idade e altura. A variável nome é uma cadeia de caracteres (*string*) que armazena o valor 'Gil Araújo', a variável idade é um número inteiro que armazena o valor 35 e a variável altura é um número de ponto flutuante (*float*) ou decimal que armazena o valor 1.75.

3.1 Regras e convenções

Tenha em mente que, ao definir variáveis em Python, os nomes devem seguir algumas regras:

- Os nomes das variáveis devem começar com uma letra (a-z, A-Z) ou um sublinhado (_).
- Os nomes das variáveis podem conter letras, dígitos (0-9) e sublinhados.
- Os nomes das variáveis são sensíveis a maiúsculas e minúsculas (case-sensitive).
- É uma boa prática escolher nomes de variáveis descritivos para facilitar a compreensão do código.

Bons exemplos:

```
>>> _codigo = "A12345B"
>>> nome_paciente = "Gil Araujo"
>>> idade_paciente = 35
>>> altura_paciente = 1.75
>>> tipo_sequenciamento = "Exoma (NGS)"
```

Um rápido contra-exemplo seria:

```
>>> cp = "A12345B"
>>> np = "Gil Araujo"
>>> ip = 35
>>> ap = 1.75
```

Note que ao definir os contra-exemplos os nomes das váriavéis não são nada informativos, prejudicando assim a legibilidade e leitura do código por terceiros.

Em geral, as variáveis são definidas como substantivos e adjetivos, enquanto que funções denotam ações e assim são geralmente definidas com verbos.

CAPÍTULO 4

Números inteiros e flutuantes

4.1 Inteiros

Em Python, números inteiros são uma forma de dados numéricos que representam valores inteiros sem parte fracionária. Eles são usados para realizar operações matemáticas simples e complexas, além de serem armazenados em variáveis para posterior utilização no código.

Os números inteiros em Python podem ser positivos, negativos ou zero. Aqui estão alguns exemplos de números inteiros:

Por exemplo:

```
1 >>> x = 1
2 >>> y = -1
3 >>> z = 0
```

Neste exemplo, **x** é um número inteiro positivo, **y** é um número inteiro negativo e **z** é o número inteiro zero.

Os números inteiros em Python têm tamanho ilimitado, o que significa que você pode trabalhar com números inteiros muito grandes sem se preocupar com limitações de tamanho. Isso é diferente de algumas outras linguagens de programação que têm um limite definido para o tamanho dos números inteiros.

4.1.1 Função int()

Em Python, a função int() é uma função embutida que converte um valor para o tipo de dado inteiro. Ela pode ser usada para converter diferentes tipos de dados em números inteiros.

A sintaxe básica da função int() é a seguinte:

```
1 >>> int(x)
```

Neste exemplo, **x** representa o valor que você deseja converter em um número inteiro, que pode ser um número ou uma string.

A função int() pode ser usada de diferentes maneiras, dependendo do tipo de dado que você está convertendo. Aqui estão alguns exemplos:

Converter uma string em um número inteiro:

```
1 >>> numero_reads = int("1000")
2 >>> print(numero_reads) # Saída:  1000
```

Nesse exemplo, a função **int()** converte a *string* "1000"em um número inteiro 1000. Isso permite que você realize operações matemáticas com o valor convertido.

NOTA: É importante notar que, ao converter uma string em um número inteiro, a string deve conter apenas dígitos (0-9) e, opcionalmente, um sinal de positivo (+) ou negativo (-) no início. Utilize a função `isdigit()` para verificar essa condição.

Converter um número de ponto flutuante em um número inteiro:

```
1 >>> valor_pi = int(3.14)
2 >>> print(valor_pi)  # Saída:  3
```

No exemplo acima, a função **int()** converte o número de ponto flutuante 3.14 em um número inteiro 3. A conversão de um número de ponto flutuante para um número inteiro trunca a parte fracionária, descartando qualquer informação após o ponto decimal.

4.2 Números flutuantes ou decimais (float)

Os números flutuantes são representados pelo tipo de dado float e assim como os inteiros é possível converter strings e inteiros em números flutuantes com a função **float()**. Eles são usados para representar valores com casas decimais. Por exemplo:

```
>>> x = 1.5
>>> y = 0.0
>>> z = -1.5
>>> # convertendo string em número
>>> v = float("1.5")
>>> u = float("-1.5")
```

NOTA: Esteja sempre iente de que operações com números de ponto flutuante podem resultar em imprecisões devido à forma como os números são armazenados e manipulados internamente pelo computador. Portanto, pode ser necessário arredondar os resultados, se necessário, usando a funções ou formatar a saída de acordo com as necessidades do seu programa.

4.3 Operações com números

Você pode realizar diversas operações com números. Além das operações aritméticas básicas (adição, subtração, multiplicação e divisão), existem outras operações matemáticas disponíveis. Segue abaixo os operadores mais utilizados em operações matemáticas de números inteiros e flutuantes:

Soma:

```
>>> 3+2 #soma
5
```

Subtração:

```
>>> 7-3 #subtracao
4
```

Multiplicação:

```
>>> 7*2 #multiplicacao
14
```

Potenciação: utiliza o operador ** para elevar um número a uma potência.

```
>>> 5**2 #exponenciacao
25
```

Divisão:

```
>>> 8/4 #divisao
2
```

Divisão inteira: utiliza o operador `//` para realizar a divisão inteira, retornando apenas a parte inteira do resultado.

```
>>> 7//2 #divisao inteira
3
```

Resto da divisão (módulo): utiliza o operador `%` para obter o resto da divisão entre dois números.

```
>>> 15 % 4
3
```

Arrendondamento:

```
>>> round(1.3)
1
>>> round(1.5)
2
```

Potenciação:

```
>>> pow(2, 4)
16
```

Conversão pra números decimaisL

```
>>> float(2)
2.0
```

Valor absoluto:

```
>>> abs(-2)
2
```

4.3.1 Módulo math

O módulo *math* em Python fornece várias funções matemáticas. Aqui estão algumas das principais funções disponíveis:

Funções trigonométricas:

- math.sin(x): retorna o seno de x (em radianos).
- math.cos(x): retorna o cosseno de x (em radianos).
- math.tan(x): retorna a tangente de x (em radianos).

Funções exponenciais e logarítmicas:

- math.exp(x): retorna a exponencial de x (eˆx).
- math.log(x): retorna o logaritmo natural (base e) de x.
- math.log10(x): retorna o logaritmo de base 10 de x.
- math.log2(x): retorna o logaritmo de base 2 de x.
- math.pow(x, y): retorna x elevado à potência y.
- math.sqrt(x): retorna a raiz quadrada de x.

Funções de arredondamento:

- math.ceil(x): retorna o menor inteiro maior ou igual a x.
- math.floor(x): retorna o maior inteiro menor ou igual a x.
- math.trunc(x): retorna a parte inteira de x.
- round(x): arredonda x para o número inteiro mais próximo.

Constantes:

- math.pi: a constante matemática π (pi).
- math.e: a constante matemática e (base do logaritmo natural).

Essas são apenas algumas das funções disponíveis na biblioteca 'math' do Python. Existem outras funções e constantes matemáticas que você pode explorar consultando a documentação oficial do Python ou experimentando a biblioteca em seu código.

CAPÍTULO 5

Strings

Uma string pode ser definida como uma sequência de caracteres delimitada por aspas simples (‘ ’) ou aspas duplas (“ ”). As strings são objetos imutáveis e não podem ser modificadas depois de criadas. Entretanto, você pode realizar diversas operações para manipulá-las ou extrair informações.

5.1 Criando *strings*

Para criar uma string é simple. Você deve atribuir um valor entre aspas a uma variável. Por exemplo:

```
1 >>> mensagem = "áOl, programador!!"
2 >>> print(mensagem)
3 "áOl, programador!!"
```

5.2 Indexação

Em Python, você pode acessar caracteres individuais de uma string usando indexação. Os índices começam em **0** para o primeiro caractere e vão até o comprimento da string menos um (**-1**). Aqui, usamos colchetes [] para acessar caracteres/posições individuais com base em seus índices. Por exemplo:

```
>>> primeiro_caractere = minha_string[0]  # Saída 'O'
>>> segundo_caractere = minha_string[1]  # Saída 'l'
>>> ultimo_caractere = minha_string[-1]  # Saída '!'
```

A indexação de strings em Python pode ser aplicada a sequências de DNA, RNA ou proteínas da mesma forma que a indexação de strings comuns. Por exemplo:

Acessar o primeiro nucleotídeo de uma sequência de DNA:

```
dna = "ATCGATCGATCG"
primeiro_nucleotideo = dna[0]
print(primeiro_nucleotideo)  # Saída:  A
```

Acessar o último aminoácido de uma sequência de proteína:

```
proteina = "MIPTEKLSR"
ultimo_aminoacido = proteina[-1]
print(ultimo_aminoacido)  # Saída:  R
```

Acessar o segundo códon:

```
dna = "ATCGATCGATCG"
codon = dna[3:6]
print(codon)  # Saída:  GAT
```

5.3 Métodos

Uma série de métodos úteis pode ser conferida na lista seguinte:

- `capitalize()`: Retorna uma cópia da string com o primeiro caractere em maiúscula e os demais em minúscula.
- `casefold()`: Retorna uma versão em minúsculas da string, adequada para comparação de strings que ignoram diferenças de caso.
- `count(substring, start, end)`: Retorna o número de ocorrências de uma substring na string.
- `endswith(suffix, start, end)`: Verifica se a string termina com uma determinada substring e retorna `True` ou `False`.
- `find(substring, start, end)`: Retorna o índice da primeira ocorrência de uma substring na string, ou -1 se não for encontrada.

- `index(substring, start, end)`: Retorna o índice da primeira ocorrência de uma substring na string, ou gera uma exceção `ValueError` se não for encontrada.
- `isalnum()`: Verifica se a string contém apenas caracteres alfanuméricos e retorna `True` ou `False`.
- `isalpha()`: Verifica se a string contém apenas caracteres alfabéticos e retorna `True` ou `False`.
- `isdecimal()`: Verifica se a string contém apenas dígitos decimais e retorna `True` ou `False`.
- `isdigit()`: Verifica se a string contém apenas dígitos e retorna `True` ou `False`.
- `isnumeric()`: Verifica se a string contém apenas caracteres numéricos e retorna `True` ou `False`.
- `upper()`: Retorna uma versão em maiúsculas da string.
- `lower()`: Retorna uma versão em minúsculas da string.
- `lstrip(characters)`: Retorna uma cópia da string com os caracteres à esquerda removidos.
- `replace(old, new, count)`: Substitui todas as ocorrências de uma substring por outra na string.
- `rfind(substring, start, end)`: Retorna o índice da última ocorrência de uma substring na string, ou -1 se não for encontrada.
- `rindex(substring, start, end)`: Retorna o índice da última ocorrência de uma substring na string, ou gera uma exceção `ValueError` se não for encontrada.
- `rstrip(characters)`: Retorna uma cópia da string com os caracteres à direita removidos.
- `split(separator, maxsplit)`: Divide a string em substrings com base em um separador e retorna uma lista.
- `startswith(prefix, start, end)`: Verifica se a string começa com um determinado prefixo e retorna `True` ou `False`.

- `strip(characters)`: Retorna uma cópia da string com os caracteres à esquerda e à direita removidos.
- `join(iterable)`: Concatena os elementos de um iterável em uma string, usando a string como separador.
- `title()`: Retorna uma versão da string com a primeira letra de cada palavra em maiúscula.

Essa lista inclui apenas alguns dos métodos disponíveis para strings em Python. Existem outros métodos e funcionalidades que podem ser explorados na documentação oficial do Python.