Guia Completo de Python: Estruturas e Sintaxe

Introdução ao Python

Python é uma linguagem de programação de alto nível, interpretada, de propósito geral, criada por Guido van Rossum em 1991. Conhecida pela sua sintaxe clara e legível, Python enfatiza a legibilidade do código e permite que os programadores expressem conceitos complexos em menos linhas de código do que seria possível em linguagens como C++ ou Java. Essa característica, combinada com sua vasta biblioteca padrão e uma comunidade ativa, tornou Python uma das linguagens mais populares e versáteis do mundo.

Este guia tem como objetivo fornecer uma referência abrangente sobre as estruturas de dados fundamentais e a sintaxe básica do Python, servindo como um recurso valioso para consulta rápida e aprofundamento. Abordaremos desde os tipos de dados essenciais até conceitos mais avançados como Programação Orientada a Objetos e testes, sempre com exemplos práticos para facilitar o entendimento.

1. Estruturas de Dados Essenciais

As estruturas de dados são a espinha dorsal de qualquer programa, permitindo organizar e armazenar informações de maneira eficiente. Python oferece várias estruturas de dados embutidas, cada uma com suas próprias características e casos de uso.

1.1. Listas (`list`)

Listas são coleções ordenadas, mutáveis e que permitem elementos duplicados. São definidas por colchetes `[]`.

```
 # Exemplo de criação de lista
frutas = ["maçã", "banana", "cereja"]
print(f"Lista inicial: {frutas}")

# Acessando elementos
print(f"Primeira fruta: {frutas[0]}")
```

```
# Modificando elementos
frutas[1] = "kiwi"
print(f"Lista após modificação: {frutas}")

# Adicionando elementos
frutas.append("laranja")
print(f"Lista após adicionar: {frutas}")

# Removendo elementos
frutas.remove("maçã")
print(f"Lista após remover: {frutas}")

# Fatiamento (slicing)
print(f"Fatiamento [0:2]: {frutas[0:2]}")
```

1.2. Tuplas (`tuple`)

Tuplas são coleções ordenadas, imutáveis e que permitem elementos duplicados. São definidas por parênteses `()`.

```
# Exemplo de criação de tupla
coordenadas = (10, 20, 30)
print(f"Tupla inicial: {coordenadas}")

# Acessando elementos
print(f"Primeira coordenada: {coordenadas[0]}")

# Tuplas são imutáveis (a linha abaixo geraria um erro)
# coordenadas[0] = 5

# Desempacotamento de tuplas
x, y, z = coordenadas
print(f"x={x}, y={y}, z={z}")
```

1.3. Dicionários (`dict`)

Dicionários são coleções não ordenadas de pares chave-valor, mutáveis e indexadas por chaves únicas. São definidos por chaves `{}`.

```
# Exemplo de criação de dicionário
pessoa = {
    "nome": "Alice",
    "idade": 30,
    "cidade": "São Paulo"
}
print(f"Dicionário inicial: {pessoa}")
```

```python
# Acessando valores por chave
print(f"Nome: {pessoa["nome"]}")

# Adicionando ou modificando pares chave-valor
pessoa["profissao"] = "Engenheira"
print(f"Dicionário após adicionar: {pessoa}")

pessoa["idade"] = 31
print(f"Dicionário após modificar: {pessoa}")

# Removendo pares chave-valor
del pessoa["cidade"]
print(f"Dicionário após remover: {pessoa}")

# Iterando sobre dicionário
print("Itens do dicionário:")
for chave, valor in pessoa.items():
    print(f"  {chave}: {valor}")
```

1.4. Conjuntos (`set`)

Conjuntos são coleções não ordenadas de elementos únicos e imutáveis. São definidos por chaves `{}` ou pela função `set()`.

```python
 # Exemplo de criação de conjunto
numeros = {1, 2, 3, 2, 4}
print(f"Conjunto inicial: {numeros}")
# Elementos duplicados são removidos automaticamente

# Adicionando elementos
numeros.add(5)
print(f"Conjunto após adicionar: {numeros}")

# Removendo elementos
numeros.remove(1)
print(f"Conjunto após remover: {numeros}")

# Operações de conjunto
outros_numeros = {3, 4, 5, 6}
print(f"União: {numeros.union(outros_numeros)}")
print(f"Interseção: {numeros.intersection(outros_numeros)}")
```

2. Lógica e Operadores

A lógica de programação é a base para construir algoritmos e tomar decisões em seu código. Python oferece uma variedade de operadores para realizar cálculos, comparações e operações lógicas.

2.1. Operadores Aritméticos

Usados para realizar operações matemáticas.

Operador	Descrição	Exemplo	Resultado
`+`	Adição	`5 + 3`	`8`
`-`	Subtração	`5 - 3`	`2`
`*`	Multiplicação	`5 * 3`	`15`
`/`	Divisão (float)	`5 / 3`	`1.66...`
`//`	Divisão (inteira)	`5 // 3`	`1`
`%`	Módulo (resto)	`5 % 3`	`2`
`**`	Exponenciação	`5 ** 3`	`125`

```
 a = 10
b = 3
print(f"Adição: {a + b}")
print(f"Subtração: {a - b}")
print(f"Multiplicação: {a * b}")
print(f"Divisão: {a / b}")
print(f"Divisão Inteira: {a // b}")
print(f"Módulo: {a % b}")
print(f"Exponenciação: {a ** b}")
```

2.2. Operadores de Comparação

Usados para comparar dois valores, retornando `True` ou `False`.

Operador	Descrição	Exemplo	Resultado
`==`	Igual a	`5 == 3`	`False`
`!=`	Diferente de	`5 != 3`	`True`
`>`	Maior que	`5 > 3`	`True`
`<`	Menor que	`5 < 3`	`False`
`>=`	Maior ou igual a	`5 >= 3`	`True`

Operador	Descrição	Exemplo	Resultado
`<=`	Menor ou igual a	`5 <= 3`	`False`

```
x = 10
y = 20
print(f"x == y: {x == y}")
print(f"x != y: {x != y}")
print(f"x > y: {x > y}")
print(f"x < y: {x < y}")
print(f"x >= y: {x >= y}")
print(f"x <= y: {x <= y}")
```

2.3. Operadores Lógicos

Usados para combinar expressões condicionais.

Operador	Descrição	Exemplo	Resultado
`and`	Retorna `True` se ambas as condições forem `True`	`True and False`	`False`
`or`	Retorna `True` se pelo menos uma condição for `True`	`True or False`	`True`
`not`	Inverte o valor booleano	`not True`	`False`

```
idade = 25
tem_carteira = True

print(f"Pode dirigir: {idade >= 18 and tem_carteira}")

tem_passaporte = False
print(f"Pode viajar: {tem_carteira or tem_passaporte}")

esta_chovendo = False
print(f"Não está chovendo: {not esta_chovendo}")
```

2.4. Estruturas Condicionais (`if`, `elif`, `else`)

Permitem que o programa execute diferentes blocos de código com base em condições.

```
nota = 75

if nota >= 90:
    print("Excelente!")
elif nota >= 70:
    print("Aprovado!")
elif nota >= 50:
    print("Recuperação.")
else:
    print("Reprovado.")
```

2.5. Estruturas de Repetição (`for`, `while`)

Permitem executar um bloco de código repetidamente.

2.5.1. Laço `for`

Usado para iterar sobre sequências (listas, tuplas, strings, etc.) ou outros objetos iteráveis.

```
# Iterando sobre uma lista
frutas = ["maçã", "banana", "cereja"]
for fruta in frutas:
    print(fruta)

# Iterando com range()
for i in range(5): # de 0 a 4
    print(i)

# Iterando com enumerate() (índice e valor)
for indice, fruta in enumerate(frutas):
    print(f"Índice {indice}: {fruta}")
```

2.5.2. Laço `while`

Executa um bloco de código enquanto uma condição for verdadeira.

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

2.5.3. `break` e `continue`

- `break`: Termina o laço imediatamente.

- `continue`: Pula a iteração atual e vai para a próxima.

```
# Exemplo com break
for i in range(10):
    if i == 5:
        break
    print(i)

# Exemplo com continue
for i in range(10):
    if i % 2 == 0: # Se for par, pula
        continue
    print(i) # Imprime apenas números ímpares
```

3. Funções

Funções são blocos de código reutilizáveis que realizam uma tarefa específica. Elas ajudam a organizar o código, torná-lo mais legível e evitar repetições.

3.1. Definição de Funções

Funções são definidas usando a palavra-chave `def`.

```
def saudar(nome):
    """Esta função saúda a pessoa com o nome fornecido."""
    print(f"Olá, {nome}!")

# Chamando a função
saudar("Alice")
saudar("Bob")
```

3.2. Parâmetros e Argumentos

- **Parâmetros:** Nomes listados na definição da função.
- **Argumentos:** Valores passados para a função quando ela é chamada.

3.2.1. Argumentos Posicionais

São passados na ordem em que os parâmetros são definidos.

```
def somar(a, b):
    return a + b
```

```
resultado = somar(5, 3)
print(f"Soma: {resultado}")
```

3.2.2. Argumentos Nomeados (Keyword Arguments)

Passados usando o nome do parâmetro, permitindo qualquer ordem.

```
def descrever_carro(marca, modelo, ano):
    print(f"Carro: {marca} {modelo}, Ano: {ano}")

descrever_carro(modelo="Civic", ano=2020, marca="Honda")
```

3.2.3. Valores Padrão para Parâmetros

Permitem que um parâmetro tenha um valor predefinido se nenhum argumento for fornecido.

```
def saudar_padrao(nome="Visitante"):
    print(f"Olá, {nome}!")

saudar_padrao()
saudar_padrao("Maria")
```

3.3. Retorno de Valores

Funções podem retornar valores usando a palavra-chave `return`.

```
def multiplicar(x, y):
    return x * y

resultado = multiplicar(4, 6)
print(f"Multiplicação: {resultado}")

def obter_info_usuario():
    nome = input("Digite seu nome: ")
    idade = int(input("Digite sua idade: "))
    return nome, idade # Retorna múltiplos valores como uma
tupla

nome_usuario, idade_usuario = obter_info_usuario()
print(f"Usuário: {nome_usuario}, Idade: {idade_usuario}")
```

3.4. Escopo de Variáveis (Local e Global)

- **Variáveis Locais:** Definidas dentro de uma função, acessíveis apenas dentro dela.
- **Variáveis Globais:** Definidas fora de qualquer função, acessíveis em todo o programa.

```
variavel_global = "Eu sou global"

def minha_funcao():
    variavel_local = "Eu sou local"
    print(variavel_local)
    print(variavel_global) # Acessando variável global

minha_funcao()
# print(variavel_local) # Isso geraria um NameError
print(variavel_global)

def modificar_global():
    global variavel_global # Declara que estamos usando a
variável global
    variavel_global = "Eu fui modificada globalmente"

modificar_global()
print(variavel_global)
```

3.5. Funções Anônimas (`lambda`)

Funções `lambda` são pequenas funções anônimas (sem nome) que podem ter qualquer número de argumentos, mas apenas uma expressão. São úteis para operações simples e rápidas.

```
somar_lambda = lambda a, b: a + b
print(f"Soma com lambda: {somar_lambda(2, 3)}")

# Usando lambda com funções de ordem superior (map, filter,
sorted)
lista_numeros = [1, 2, 3, 4, 5]
quadrados = list(map(lambda x: x**2, lista_numeros))
print(f"Quadrados: {quadrados}")

pare = list(filter(lambda x: x % 2 == 0, lista_numeros))
print(f"Pares: {pare}")
```

4. Programação Orientada a Objetos (POO)

Programação Orientada a Objetos (POO) é um paradigma de programação que organiza o design do software em torno de objetos, em vez de funções e lógica. Em Python, tudo é um objeto, o que torna a POO uma parte natural da linguagem.

4.1. Classes e Objetos

- **Classe:** Um blueprint ou modelo para criar objetos. Define um conjunto de atributos (variáveis) e métodos (funções) que os objetos criados a partir dela terão.
- **Objeto (Instância):** Uma ocorrência concreta de uma classe.

```
class Carro:
    # Atributo de classe (compartilhado por todas as instâncias)
    rodas = 4

    def __init__(self, marca, modelo, cor): # Método construtor
        self.marca = marca   # Atributo de instância
        self.modelo = modelo # Atributo de instância
        self.cor = cor       # Atributo de instância
        self.velocidade = 0

    def acelerar(self, incremento):
        self.velocidade += incremento
        print(f"{self.modelo} acelerou para {self.velocidade}
km/h.")

    def frear(self, decremento):
        self.velocidade -= decremento
        print(f"{self.modelo} freou para {self.velocidade} km/
h.")

    def exibir_detalhes(self):
        print(f"Marca: {self.marca}, Modelo:
{self.modelo}, Cor: {self.cor}, Rodas: {Carro.rodas}")

# Criando objetos (instâncias da classe Carro)
meu_carro = Carro("Toyota", "Corolla", "Prata")
carro_amigo = Carro("Honda", "Civic", "Preto")

# Acessando atributos
print(f"Meu carro é um {meu_carro.marca} {meu_carro.modelo}.")

# Chamando métodos
meu_carro.acelerar(50)
carro_amigo.exibir_detalhes()
```

4.2. Encapsulamento

Encapsulamento é o princípio de agrupar dados (atributos) e os métodos que operam nesses dados dentro de uma única unidade (classe), e restringir o acesso direto a alguns dos componentes do objeto. Em Python, o encapsulamento é mais uma convenção do que uma imposição rígida.

- **Atributos Públicos:** Acessíveis diretamente de fora da classe (convenção: `nome_atributo`).
- **Atributos Protegidos:** Indicados por um único underscore `_` (convenção: `_nome_atributo`). Sugere que o atributo não deve ser acessado diretamente de fora da classe, mas não impede.
- **Atributos Privados:** Indicados por dois underscores `__` (convenção: `__nome_atributo`). Python faz um "name mangling" para dificultar o acesso direto, mas ainda é possível.

```
class ContaBancaria:
    def __init__(self, titular, saldo_inicial):
        self.titular = titular       # Público
        self._numero_conta = "12345" # Protegido (convenção)
        self.__saldo = saldo_inicial # Privado (name mangling)

    def depositar(self, valor):
        if valor > 0:
            self.__saldo += valor
            print(f"Depósito de R${valor:.2f} realizado. Novo
saldo: R${self.__saldo:.2f}")
        else:
            print("Valor de depósito inválido.")

    def sacar(self, valor):
        if 0 < valor <= self.__saldo:
            self.__saldo -= valor
            print(f"Saque de R${valor:.2f} realizado. Novo
saldo: R${self.__saldo:.2f}")
        else:
            print("Saldo insuficiente ou valor de saque
inválido.")

    def get_saldo(self): # Método para acessar o saldo privado
        return self.__saldo

# Uso da classe
minha_conta = ContaBancaria("João", 1000)
# print(minha_conta.__saldo) # Erro: AttributeError
print(f"Saldo atual: R${minha_conta.get_saldo():.2f}")
```

```
minha_conta.depositar(200)
minha_conta.sacar(1500)
```

4.3. Herança

Herança é um mecanismo que permite que uma nova classe (subclasse/classe filha) herde atributos e métodos de uma classe existente (superclasse/classe pai). Isso promove a reutilização de código e estabelece uma relação "é um tipo de".

Sintaxe da Herança:

```
class Superclasse:
    # Atributos e métodos da superclasse
    pass

class Subclasse(Superclasse):
    # Atributos e métodos da subclasse, além dos herdados
    pass
```

Exemplo:

```
class Animal:
    def __init__(self, nome):
        self.nome = nome

    def comer(self):
        print(f"{self.nome} está comendo.")

class Cachorro(Animal):
    def __init__(self, nome, raca):
        super().__init__(nome) # Chama o construtor da
superclasse
        self.raca = raca

    def latir(self):
        print(f"{self.nome} ({self.raca}) está latindo.")

# Criando objetos
meu_animal = Animal("Bichano")
meu_cachorro = Cachorro("Rex", "Labrador")

meu_animal.comer()
meu_cachorro.comer() # Método herdado
meu_cachorro.latir()
```

4.4. Polimorfismo

Polimorfismo significa "muitas formas". Em POO, refere-se à capacidade de objetos de diferentes classes responderem ao mesmo método de maneiras diferentes. Isso é frequentemente alcançado através da sobrescrita de métodos (method overriding).

```
class Forma:
    def area(self):
        raise NotImplementedError("Subclasses devem implementar
este método")

class Retangulo(Forma):
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    def area(self):
        return self.largura * self.altura

class Circulo(Forma):
    def __init__(self, raio):
        self.raio = raio

    def area(self):
        import math
        return math.pi * (self.raio ** 2)

# Função polimórfica
def imprimir_area(forma):
    print(f"A área da forma é: {forma.area():.2f}")

ret = Retangulo(5, 10)
circ = Circulo(7)

imprimir_area(ret)
imprimir_area(circ)
```

5. Debugging e Testes

Debugging e testes são etapas cruciais no desenvolvimento de software para garantir que o código funcione como esperado e esteja livre de erros. Uma abordagem sistemática para depuração e a escrita de testes automatizados são práticas essenciais para qualquer desenvolvedor.

5.1. Debugging

Debugging é o processo de encontrar e resolver bugs (erros) no código. Python oferece ferramentas e técnicas para auxiliar nesse processo.

5.1.1. Impressão de Mensagens (`print()`)

A forma mais simples e comum de depurar é usar a função `print()` para exibir o valor de variáveis, o fluxo de execução do programa e mensagens de status.

```
def calcular_media(lista_numeros):
    print(f"DEBUG: Lista recebida: {lista_numeros}")
    if not lista_numeros:
        print("DEBUG: Lista vazia, retornando 0.")
        return 0
    soma = sum(lista_numeros)
    print(f"DEBUG: Soma dos números: {soma}")
    media = soma / len(lista_numeros)
    print(f"DEBUG: Média calculada: {media}")
    return media

print(calcular_media([10, 20, 30]))
print(calcular_media([]))
```

5.1.2. Módulo `pdb` (Python Debugger)

O `pdb` é o depurador interativo padrão do Python. Ele permite pausar a execução do programa, inspecionar variáveis, executar código linha por linha e muito mais.

- **Iniciando o `pdb`:**

  Você pode iniciar o `pdb` de várias maneiras:

  - **No início do script:** `python import pdb pdb.set_trace() # Seu código começa aqui`
  - **Em um ponto específico:** `python # Seu código if alguma_condicao: import pdb pdb.set_trace() # Mais código`
  - **Executando o script com `pdb`:** `bash python -m pdb seu_script.py`

- **Comandos Comuns do `pdb`:**

Comando	Descrição
`n` (next)	Executa a próxima linha de código. Se a linha for uma chamada de função, executa a função inteira.

Comando	Descrição
`s` (step)	Executa a próxima linha de código. Se a linha for uma chamada de função, entra na função.
`c` (continue)	Continua a execução até o próximo breakpoint ou o fim do programa.
`q` (quit)	Sai do depurador.
`p <expressão>`	Imprime o valor de uma expressão ou variável.
`l` (list)	Lista o código-fonte ao redor da linha atual.
`b <linha>`	Define um breakpoint em uma linha específica.
`cl` (clear)	Limpa todos os breakpoints ou um breakpoint específico.
`w` (where)	Exibe o stack trace atual.

5.2. Testes Automatizados

Testes automatizados são códigos escritos para verificar se outras partes do seu código funcionam conforme o esperado. Eles são essenciais para garantir a qualidade do software, facilitar a refatoração e prevenir regressões.

5.2.1. Módulo `unittest`

O `unittest` é o framework de teste de unidade padrão do Python, inspirado no JUnit do Java.

- **Estrutura Básica de um Teste:**

  Um teste com `unittest` geralmente envolve: * Importar `unittest`. * Criar uma classe de teste que herda de `unittest.TestCase`. * Definir métodos de teste que começam com `test_`. * Usar métodos de asserção (ex: `assertEqual`, `assertTrue`, `assertRaises`) para verificar resultados.

```
 # arquivo: calculadora.py
def somar(a, b):
    return a + b

def subtrair(a, b):
```

```
    return a - b

# arquivo: test_calculadora.py
import unittest
from calculadora import somar, subtrair

class TestCalculadora(unittest.TestCase):

    def test_somar(self):
        self.assertEqual(somar(2, 3), 5)
        self.assertEqual(somar(-1, 1), 0)
        self.assertEqual(somar(-1, -1), -2)

    def test_subtrair(self):
        self.assertEqual(subtrair(5, 2), 3)
        self.assertEqual(subtrair(2, 5), -3)
        self.assertEqual(subtrair(0, 0), 0)

if __name__ == '__main__':
    unittest.main()
```

- **Executando Testes:**

  `bash python -m unittest test_calculadora.py`

5.2.2. Módulo `pytest`

`pytest` é um framework de teste popular e mais moderno que o `unittest`, conhecido por sua sintaxe mais simples e recursos poderosos. Ele não exige que as classes de teste herdem de uma classe base específica.

- **Instalação:**

  `bash pip install pytest`

- **Estrutura Básica de um Teste com `pytest`:**
  - Funções de teste começam com `test_`.
  - Usa a palavra-chave `assert` para verificações.

```
 # arquivo: calculadora.py (mesmo do exemplo unittest)
def somar(a, b):
    return a + b

def subtrair(a, b):
    return a - b

# arquivo: test_calculadora_pytest.py
```

```python
from calculadora import somar, subtrair

def test_somar():
    assert somar(2, 3) == 5
    assert somar(-1, 1) == 0
    assert somar(-1, -1) == -2

def test_subtrair():
    assert subtrair(5, 2) == 3
    assert subtrair(2, 5) == -3
    assert subtrair(0, 0) == 0
```

- **Executando Testes com `pytest`:**

  ```bash
  pytest
  ```

5.2.3. Test-Driven Development (TDD)

TDD é uma metodologia de desenvolvimento de software onde os testes são escritos antes do código de produção. O ciclo TDD consiste em:

1. **Escrever um teste que falha:** Comece escrevendo um teste para uma nova funcionalidade que ainda não existe, então ele falhará.
2. **Escrever o código mínimo para o teste passar:** Implemente apenas o código necessário para que o teste recém-escrito passe.
3. **Refatorar o código:** Melhore a estrutura e a qualidade do código, garantindo que todos os testes continuem passando.

Este ciclo se repete, garantindo que cada nova funcionalidade seja coberta por testes e que o código permaneça limpo e funcional.

5.3. Boas Práticas de Teste

- **Testes de Unidade:** Teste pequenas partes isoladas do código (funções, métodos).
- **Testes de Integração:** Teste como diferentes partes do sistema interagem entre si.
- **Testes de Aceitação:** Verifique se o sistema atende aos requisitos do usuário.
- **Testes Rápidos:** Testes devem ser rápidos para serem executados frequentemente.
- **Testes Independentes:** Cada teste deve ser independente dos outros.
- **Cobertura de Código:** Busque uma boa cobertura de código, mas não se prenda a 100% como única métrica.
- **Mocks e Stubs:** Use mocks e stubs para isolar dependências externas durante os testes.

Debugging e testes são habilidades complementares que, quando dominadas, elevam significativamente a qualidade e a confiabilidade do seu código Python.