

Linguagem de Programação

Introdução à Linguagem Python

Prof.ª Elisa Antolli

- Unidade de Ensino: 01
- Competência da Unidade: Conhecer a linguagem de programação Python.
- Resumo: Saber utilizar modelos de programação na linguagem Python.
- Palavras-chave: Linguagem de programação; Python;
- Programação; Desenvolvimento; Algoritmos.
- Título da Teleaula: Introdução a linguagem Python
- Teleaula nº: 01

Principais conceitos de programação em Python

- A linguagem Python
- Ferramentas
- Estruturas lógicas, condicionais e de repetição
- Funções
- Exemplos de códigos

Introdução linguagem de Programação Python

O que é Linguagem de Programação?

- As linguagens de programação foram criadas para solucionar qualquer tipo de problema na área tecnológica computacional.
- Cada linguagem possui suas particularidades.
- Permite que um programador crie programas a partir de um conjunto de ordens, ações consecutivas, dados e algoritmos.
- Python é uma linguagem de script de alto nível, de tipagem forte e dinâmica

Primeiros passos em Python.

- Vamos criar nosso primeiro programa em Python.



A Linguagem Python

- Criado no início dos anos 1990 por Guido van Rossum no Stichting Mathematisch Centrum (CWI), na Holanda, foi sucessor de uma linguagem chamada ABC.
- Em 2001, a Python Software Foundation (PSF) foi formada, uma organização sem fins lucrativos criada especificamente para possuir a propriedade intelectual relacionada ao Python.

A Linguagem Python

Porque Python?

- Python é uma linguagem de programação clara e poderosa.
- Usa sintaxe clara, facilitando a leitura dos programas que você escreve;
- Linguagem fácil, ideal para o desenvolvimento de protótipos e outras tarefas de programação;
- Grande biblioteca padrão, suporta muitas tarefas de programação;
- Possui inúmeras bibliotecas que estendem seu poder de atuação.

A Linguagem Python

Porque Python?

- Linguagem interpretada, ou seja, uma vez escrito o código, este não precisa ser convertido em linguagem de máquina por um processo de compilação;
- Permite atribuição múltipla;
- O interpretador Python 3 utiliza unicode por padrão, o que torna possível usar nomes de variáveis com acento e até outros caracteres especiais, porém não é uma boa prática.
- Códigos em Python pode ser feito tanto em local quanto em nuvem.

A Linguagem Python

Instalação do interpretador Python:

<https://www.python.org/downloads/>

Na instalação marcar a opção Add Python 3.X to PATH.



A Linguagem Python

Já podemos digitar comandos python:

```

Microsoft Windows [versão 10.0.19044.1130]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Usuario\python
python 3.10.8 (tags/v3.10.8:aa7517, Oct 11 2022, 16:56:38) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Ola mundo!")
Ola mundo!
>>>
  
```

Fonte adaptado pelo autor

A Linguagem Python

Mais ferramentas:

- Para implementação de soluções, normalmente utiliza-se uma IDE, (Integrated Development Environment) ou Ambiente de Desenvolvimento Integrado.
- Duas IDE's disputam a preferência dos desenvolvedores Python, o PyCharm e o Visual Studio Code (VSCode).
- PyCharm: Profissional e Community, sendo a primeira paga e a segunda gratuita.
- VSCode: Gratuito

A Linguagem Python

Mais ferramentas:

- Python Anaconda (<https://www.anaconda.com/distribution/>). Consiste na união de ferramentas Python, compostas por bibliotecas e IDE's.
- Possui tanto o interpretador Python quanto bibliotecas, duas interfaces de desenvolvimento: a IDE spyder e o projeto Jupyter.
- Grande diferencial do projeto Anaconda é ter o Jupyter Notebook (<https://jupyter.org/>) integrado na instalação, principalmente para o uso sistemas de controle de versão (como git / GitHub).

A Linguagem Python

Mais ferramentas :

Google Colaboratory (Colab)

<https://colab.research.google.com/notebooks/>

- Especialmente adequado para aprendizado de máquina, análise de dados e educação. Colab é um serviço de notebook Jupyter hospedado que não requer configuração para ser usado.

Variáveis e tipos básicos de dados em Python

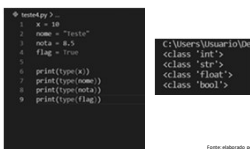
Variáveis

- Variáveis são espaços alocados na memória RAM para guardar valores temporariamente.
- Em Python, esses espaços não precisam ser tipados, a variável pode ser alocada sem especificar o tipo de dado que ela aguardará.
- As variáveis são tipadas dinamicamente nessa linguagem

Variáveis

Veja alguns exemplos:

- Para saber o tipo de dado que uma variável guarda, podemos imprimir seu tipo usando a função `type()`, veja como:



```

# teste.py
1 x = 10
2 nome = "teste"
3 nota = 8.5
4 flag = True
5
6 print(type(x))
7 print(type(nome))
8 print(type(nota))
9 print(type(flag))

```

```

C:\Users\Usuario>python teste.py
<class 'int'>
<class 'str'>
<class 'float'>
<class 'bool'>

```

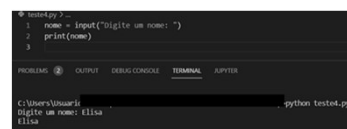
Fonte: elaborado pelo autor.

Variáveis

Em Python, tudo é objeto! Por isso os tipos de dados aparecem com a palavra "class".

Função `input()` faz a leitura de um valor digitado.

Veja como usar:



```

# teste.py
1 nome = input("Digite um nome: ")
2 print(nome)
3

```

PROBLEMA OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```

C:\Users\Usuario>python teste4.py
Digite um nome: Elisa
Elisa

```

Fonte: elaborado pelo autor.

1. Primeiro resolvem-se os parênteses, do mais interno para o mais externo.
2. Exponenciação.
3. Multiplicação e divisão.
4. Soma e subtração.

Operações matemáticas suportadas por Python

```
# Qual o resultado armazenado na variável operacao_1: 25 ou 17?
operacao_1 = 2 + 3 * 5
# Qual o resultado armazenado na variável operacao_2: 25 ou 17?
operacao_2 = (2 + 3) * 5
# Qual o resultado armazenado na variável operacao_3: 4 ou 17?
operacao_3 = 4 / 2 ** 2
# Qual o resultado armazenado na variável operacao_4: 1 ou 5?
operacao_4 = 13 % 3 + 4
```

```
Resultado em operacao_1 = 17
Resultado em operacao_2 = 25
Resultado em operacao_3 = 1.0
Resultado em operacao_4 = 5
```

Fonte: elaborado pelo autor.

Estruturas Lógicas, Condicionais e de Repetição em Python

Estruturas Lógicas, Condicionais e de Repetição em Python

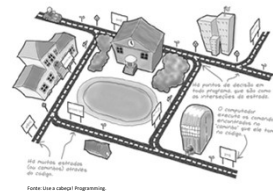
Em geral, em um programa você tem opções de caminhos ou lista de comandos que nada mais são que trechos de códigos que podem ser executados, devendo-se tomar decisões sobre qual trecho de código será executado em um determinado momento.



Fonte: Didática Tech.

Estruturas Lógicas, Condicionais e de Repetição em Python

Pontos de decisões:



Fonte: Livro a Ciência da Programação.

Estruturas Lógicas, Condicionais e de Repetição em Python

Para tomarmos decisões, precisamos dos operadores relacionais:

Operador	Descrição
==	Igual
!=	Não igual
>	Maior que
<	Menor que
>=	Maior ou igual que
<=	Menor ou igual que

Fonte: python.org

Estruturas Lógicas, Condicionais e de Repetição em Python

O comando if.. else.. significam se.. senão.. e são usados para construir as estruturas condicionais.



Fonte: Livro a Ciência da Programação.

Estruturas Lógicas, Condicionais e de Repetição em Python

Estrutura condicional simples:

```
nome = 'Daniel'
sobrenome = ''
lista = []

if nome:
    print('A variável nome não é vazia')
```

Fonte: <https://www.dbooks.com>.

Estruturas Lógicas, Condicionais e de Repetição em Python

Estrutura composta:

```
valor1 = 10
valor2 = 20

if valor1 > valor2:
    print('O valor1 é maior do que o valor2')
else:
    print('O valor2 é maior do que o valor1')
```

Fonte: <https://www.dbooks.com>.

Estruturas Lógicas, Condicionais e de Repetição em Python

Estrutura encadeada, devemos usar o comando "elif", que é uma abreviação de else if.

```
cor = "alguma cor"

if cor == 'verde':
    print('Acelerar')
elif cor == 'amarelo':
    print('Atenção')
else:
    print('Parar')
```

Fonte: <https://www.dbooks.com>.

Estruturas Lógicas, Condicionais e de Repetição em Python

Estruturas lógicas em Python: **and, or, not**

Podemos usar os operadores booleanos para construir estruturas de decisões mais complexas.

Operador booleano **and**: o resultado será True, quando os dois argumentos forem verdadeiros.

Operador booleano **or**: o resultado será True, quando pelo menos um dos argumentos for verdadeiro.

Operador booleano **not**: ele irá inverter o valor do argumento.

Portanto, se o argumento for verdadeiro, a operação o transformará em falso e vice-versa.

Estruturas Lógicas, Condicionais e de Repetição em Python

Exemplo:

Estrutura condicional usando os operadores booleanos. Um aluno só pode ser aprovado caso ele tenha menos de 5 faltas e média final igual ou superior a 7.

```
qtde_faltas = int(input("Digite a quantidade de faltas: "))
media_final = float(input("Digite a média final: "))

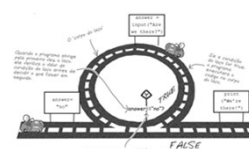
if qtde_faltas <= 5 and media_final >= 7:
    print("Aluno aprovado!")
else:
    print("Aluno reprovado!")
```

Fonte: elaborado pelo autor.

Estruturas Lógicas, Condicionais e de Repetição em Python

Estruturas de repetição em Python: **while** e **for**

Em uma estrutura de repetição sempre haverá uma estrutura decisão, pois a repetição de um trecho de código sempre está associada a uma condição. Ou seja, um bloco de comandos será executado repetidas vezes, até que uma condição não seja mais satisfeita.



Fonte: livro a coleção Programming.

Estruturas Lógicas, Condicionais e de Repetição em Python

O comando **while** deve ser utilizado para construir e controlar a estrutura decisão, sempre que o número de repetições não seja conhecido.

```
numero = 1
while numero != 0:
    numero = int(input("Digite um número: "))
    if numero % 2 == 0:
        print("Número par!")
    else:
        print("Número ímpar!")
```

Fuente: elaborado pelo autor.

Todo o bloco com a indentação de uma tabulação (4 espaços) faz parte da estrutura de repetição. Lembre: todos os blocos de comandos em Python são controlados pela indentação.

Estruturas Lógicas, Condicionais e de Repetição em Python

Na prática é comum utilizarmos esse tipo de estrutura de repetição, com **while**, para deixarmos serviços executando em servidores.

A instrução Python **for** itera sobre os itens de qualquer sequência, por exemplo, iterar sobre os caracteres de uma palavra, pois uma palavra é um tipo de sequência

Estruturas Lógicas, Condicionais e de Repetição em Python

O comando **for** seguido da variável de controle "c", na sequência o comando "in", por fim, a sequência sobre a qual a estrutura deve iterar. Os dois pontos marcam o início do bloco que deve ser repetido.

```
nome = "Guido"
for c in nome:
    print(c)
```

Fuente: elaborado pelo autor.

Estruturas Lógicas, Condicionais e de Repetição em Python

Com o comando **for**, podemos usar a função `enumerate()` para retornar à posição de cada item, dentro da sequência.

Considerando o exemplo dado, no qual atribuímos a variável "nome" o valor de "Guido", "G" ocupa a posição 0 na sequência, "u" ocupa a posição 1, "i" a posição 2, e assim por diante. Veja que a variável "i" é usada para capturar a posição e a variável "c" cada caractere da palavra.

```
nome = "Guido"
for i, c in enumerate(nome):
    print(f"Posição = {i}, valor = {c}")
```

Fuente: elaborado pelo autor.

Estruturas Lógicas, Condicionais e de Repetição em Python

Controle de repetição com **range**, **break** e **continue**:

Python requer uma sequência para que ocorra a iteração. Para criar uma sequência numérica de iteração em Python, podemos usar a função `range()`.

```
for x in range(5):
    print(x)
```

Fuente: elaborado pelo autor.

No comando, "x" é a variável de controle, ou seja, a cada iteração do laço, seu valor é alterado, já a função `range()` foi utilizada para criar um "iterable" numérico (objeto iterável) para que as repetições acontecesse.

Estruturas Lógicas, Condicionais e de Repetição em Python

A função `range()` pode ser usada de três formas distintas:

Método 1: passando um único argumento que representa a quantidade de vezes que o laço deve repetir;

Método 2: passando dois argumentos, um que representa o início das repetições e outro o limite superior (NÃO INCLUIDO) do valor da variável de controle;

Método 3: Passando três argumentos, um que representa o início das repetições; outro, o limite superior (NÃO INCLUIDO) do valor da variável de controle e um que representa o incremento.

Estruturas Lógicas, Condicionais e de Repetição em Python

Além de controlar as iterações com o tamanho da sequência, outra forma de influenciar no fluxo é por meio dos comandos "*break*" e "*continue*".

O comando ***break*** "para" a execução de uma estrutura de repetição, já com o comando ***continue***, conseguimos "pular" algumas execuções, dependendo de uma condição.

Estruturas Lógicas, Condicionais e de Repetição em Python

```
# Exemplo de uso do break
disciplina = "Linguagem de programação"

for c in disciplina:
    if c == 'a':
        break
    else:
        print(c)

# Exemplo de uso do continue
disciplina = "Linguagem de programação"
for c in disciplina:
    if c == 'a':
        continue
    else:
        print(c)
```

Fonte: elaborado pelo autor.

Implementando Soluções em Python Mediante Funções

Implementando Soluções em Python Mediante Funções

Solução dividindo-a em funções (blocos), além de ser uma boa prática de programação, tal abordagem facilita a leitura, a manutenção e a escalabilidade da solução.

- `print()` é uma função built-in do interpretador Python

Implementando Soluções em Python Mediante Funções

Função **built-in** é um objeto que está integrado ao núcleo do interpretador, não precisa ser feita nenhuma instalação adicional.

Bibliotecas Padrão			
A	E	L	R
<code>abs()</code>	<code>enumerate()</code>	<code>len()</code>	<code>range()</code>
<code>all()</code>	<code>eval()</code>	<code>list()</code>	<code>repr()</code>
<code>any()</code>	<code>exec()</code>	<code>map()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>filter()</code>	<code>max()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>format()</code>	<code>min()</code>	<code>type()</code>
<code>bool()</code>	<code>getattr()</code>	<code>next()</code>	<code>zip()</code>
<code>bytes()</code>	<code>globals()</code>	<code>open()</code>	<code>__delattr__()</code>
<code>callable()</code>	<code>hasattr()</code>	<code>os.path()</code>	<code>__dir__()</code>
<code>chr()</code>	<code>hash()</code>	<code>os.path.abspath()</code>	<code>__doc__</code>
<code>classmethod()</code>	<code>hex()</code>	<code>os.path.exists()</code>	<code>__file__</code>
<code>complex()</code>	<code>id()</code>	<code>os.path.isfile()</code>	<code>__name__</code>
<code>delattr()</code>	<code>input()</code>	<code>os.path.join()</code>	<code>__new__()</code>
<code>dict()</code>	<code>int()</code>	<code>os.path.splitext()</code>	<code>__reduce__()</code>
<code>dir()</code>	<code>isinstance()</code>	<code>os.path.split()</code>	<code>__reduce_ex__()</code>
<code>divmod()</code>	<code>isnumeric()</code>	<code>os.path.splitext()</code>	<code>__repr__()</code>
<code>enumerate()</code>	<code>isupper()</code>	<code>os.path.splitext()</code>	<code>__setattr__()</code>
<code>eval()</code>	<code>islower()</code>	<code>os.path.splitext()</code>	<code>__sizeof__()</code>
<code>exec()</code>	<code>isalpha()</code>	<code>os.path.splitext()</code>	<code>__str__()</code>
<code>filter()</code>	<code>isspace()</code>	<code>os.path.splitext()</code>	<code>__subclasshook__()</code>
<code>format()</code>	<code>isdecimal()</code>	<code>os.path.splitext()</code>	<code>__sub__()</code>
<code>getattr()</code>	<code>isdigit()</code>	<code>os.path.splitext()</code>	<code>__truediv__()</code>
<code>globals()</code>	<code>isidentifier()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>hasattr()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>hash()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>id()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>input()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>int()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isinstance()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isnumeric()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isupper()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>islower()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isalpha()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isspace()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isdecimal()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isdigit()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isidentifier()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>iskeyword()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>islower()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isupper()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isalpha()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isspace()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isdecimal()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isdigit()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>isidentifier()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>
<code>iskeyword()</code>	<code>iskeyword()</code>	<code>os.path.splitext()</code>	<code>__xor__()</code>

Fonte: python.org

Implementando Soluções em Python Mediante Funções

A função `eval()` usada no código recebe como entrada uma string digitada pelo usuário, que nesse caso é uma equação linear.

```
x = 2
y = 1

mensagem = input("Digite a fórmula geral da equação linear (a * x + b): ")
print(f"Uma entrada de usuário (equacao) e do tipo {type(equacao)}")

for x in range(3):
    y = eval(equacao)
    print(f"Resultado da equação para x = {x} é {y}")
```

Fonte: elaborado pelo autor.

Essa entrada é analisada e avaliada como uma expressão Python pela função `eval()`. Veja que, para cada valor de `x`, a fórmula é executada como uma expressão matemática (linha 8) e retorna um valor diferente. Precaução para o uso, pois é fácil alguém externo à aplicação fazer uma "injection" de código intruso.

Implementando Soluções em Python Mediante Funções

Função definida pelo usuário

- Podemos escolher o nome da função, sua entrada e sua saída.
- Nomes das funções devem estar em minúsculas, com as palavras separadas por underline, conforme necessário, para melhorar a legibilidade.
- Os nomes de variáveis seguem a mesma convenção que os nomes de funções.
- É preciso abrir e fechar parênteses, pois é dentro dos parênteses que os parâmetros de entrada da função devem ser definidos.
- Usamos o comando "def" para indicar que vamos definir uma função. Em seguida, escolhemos o nome da função "imprimir_mensagem"

Implementando Soluções em Python Mediante Funções

Exemplo:

A função abaixo recebe dois parâmetros. Esses parâmetros são variáveis locais, ou seja, são variáveis que existem somente dentro da função.

```
def imprimir_mensagem(disciplina, curso):
    print("Minha primeira função em Python desmontada na disciplina: (disciplina), do curso: (curso).")
    imprimir_mensagem("python", "ds")
```

Fonte: elaborado pelo autor.

Implementando Soluções em Python Mediante Funções

Funções com parâmetros definidos e indefinidos

Sobre os argumentos que uma função pode receber, para nosso estudo, vamos classificar em seis grupos:

1. Parâmetro posicional, obrigatório, sem valor default (padrão), tentar invocar a função, sem passar os parâmetros, acarreta um erro.

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

r = somar(2, 3)
print(r)
```

Fonte: elaborado pelo autor.

Implementando Soluções em Python Mediante Funções

2. Parâmetro posicional, obrigatório, com valor default (padrão), quando a função for invocada, caso nenhum valor seja passado, o valor default é utilizado.

```
def calcular_desconto(valor, desconto=0):
    # o parâmetro desconto possui zero valor default
    valor_com_desconto = valor - (valor * desconto)
    return valor_com_desconto

valor1 = calcular_desconto(100) # não aplicar nenhum desconto
valor2 = calcular_desconto(100, 0.25) # Aplicar desconto de 25%
print(f"O primeiro valor a ser pago = {valor1}")
print(f"O segundo valor a ser pago = {valor2}")
```

Fonte: elaborado pelo autor.

Implementando Soluções em Python Mediante Funções

3. Parâmetro nominal, obrigatório, sem valor default (padrão). Não mais importa a posição dos parâmetros, pois eles serão identificados pelo nome, a chamada da função é obrigatório passar todos os valores e sem valor default.

```
def converter_minuscula(texto, flag_minuscula):
    if flag_minuscula:
        return texto.upper()
    else:
        return texto.lower()

texto = converter_minuscula(flag_minuscula=True, texto="João") # Passagem nominal de parâmetros
print(texto)
```

Fonte: elaborado pelo autor.

Implementando Soluções em Python Mediante Funções

4. Parâmetro nominal, obrigatório, com valor default (padrão), nesse grupo os parâmetros podem possuir valor default.

```
def converter_minuscula(texto, flag_minuscula=True): # o parâmetro flag_minuscula possui valor default
    if flag_minuscula:
        return texto.lower()
    else:
        return texto.upper()

texto1 = converter_minuscula(flag_minuscula=True, texto="Introdução de Programação")
texto2 = converter_minuscula(flag_minuscula=False, texto="Introdução de Programação")
print(f"texto 1 = {texto1}")
print(f"texto 2 = {texto2}")
```

Fonte: elaborado pelo autor.

Implementando Soluções em Python Mediante Funções

5. Parâmetro posicional e não obrigatório (args), a passagem de valores é feita de modo posicional, porém a quantidade não é conhecida.

```
def imprimir_parametros(*args):
    qtd_parametros = len(args)
    print(f"Quantidade de parametros = {qtd_parametros}")

    for i, valor in enumerate(args):
        print(f"Posição = {i}, valor = {valor}")

    print("\nchamada 1")
    imprimir_parametros("São Paulo", 10, 23.78, "João")
    print("\nchamada 2")
    imprimir_parametros(10, "São Paulo")
```

Fonte: elaborado pelo autor.

Implementando Soluções em Python Mediante Funções

6. Parâmetro nominal e não obrigatório (kwargs), agora a passagem é feita de modo nominal e não posicional, o que nos permite acessar tanto o valor do parâmetro quanto o nome da variável que o armazena.

```
def imprimir_parametros(**kwargs):
    print(f"Tipo de objeto recebido = {type(kwargs)}")
    qtd_parametros = len(kwargs)
    print(f"Quantidade de parametros = {qtd_parametros}")

    for chave, valor in kwargs.items():
        print(f"variável = {chave}, valor = {valor}")

    print("\nchamada 1")
    imprimir_parametros(cidade="São Paulo", idade=33, nome="João")
    print("\nchamada 2")
    imprimir_parametros(desconto=10, valor=100)
```

Fonte: elaborado pelo autor.

Implementando Soluções em Python Mediante Funções

Funções anônimas em Python

Uma função anônima é uma função que não é construída com o "def" e, por isso, não possui nome. Esse tipo de construção é útil, quando a função faz somente uma ação e é usada uma única vez.

Poderoso recurso da linguagem Python: a expressão "lambda".

```
somar = lambda x, y: x + y
somar(x=5, y=3)
```

Fonte: elaborado pelo autor.

Recapitulando

Recapitulando

- Introdução a linguagem Python
- Variáveis e tipos básicos de dados em Python
- Estruturas Lógicas, Condicionais e de Repetição em Python
- Implementando Soluções em Python Mediante Funções
- Importância em saber utilizar modelos de estrutura de dados.

