

1. Introdução ao Python

Objetivo: Apresentar um breve histórico, as principais características e aplicações da linguagem Python.

A linguagem Python foi concebida por Guido van Rossum no fim da década 1980, enquanto trabalhava no CWI (Centrum Wiskunde & Informatica), localizado em Amsterdam, Holanda. A primeira versão (0.9.0) foi lançada no dia 20 de fevereiro de 1991. Atualmente possui um modelo de desenvolvimento comunitário, aberto e gerenciado pela Python Software Foundation (organização sem fins lucrativos). Até este momento, a última versão disponível é a 3.8.5, lançada em 20 de julho de 2020.

Havia necessidade de uma linguagem que "preenchesse o vazio entre C e o shell". Por um tempo longo, esse foi o principal objetivo do Python. — Guido Van Rossum

Atualmente, Python é quarta linguagem mais popular, de acordo com uma pesquisa conduzida pelo site Stack Overflow em 2020. (<https://insights.stackoverflow.com/survey/2020#most-popular-technologies> - acessado em 27/08/2020)

O nome Python teve a sua origem no grupo humorístico britânico Monty Python, criador do programa Monty Python's Flying Circus.

A linguagem Python apresenta as seguintes características:

- Linguagem de programação de alto nível
- Multiparadigma: orientação a objetos, imperativo, funcional e procedural.
- Possui tipagem dinâmica
- Permite fácil leitura do código
- Apresenta código mais conciso se comparado com outras linguagens

Algumas das principais aplicações da linguagem:

- Análise de dados
- Processamento de textos
- Criação de CGIs para páginas web dinâmicas

Python não utiliza:

- ; (marcadores de fim de linha)
- { } marcadores de início e fim de bloco
- **begin/end** palavras especiais para início e fim de bloco

Indentação

Indentar (indentation, em inglês) é o recuo do texto em relação a sua margem. Por exemplo, se antes de escrever uma instrução, utiliza-se 4 espaçamentos da margem esquerda até a instrução propriamente dita, a indentação utilizada possui 4 espaços.

Em Python, a indentação possui função muito importante. Os blocos de instrução são delimitados pela profundidade da indentação. As instruções que estiverem rente a margem esquerda, fazem parte do primeiro nível hierárquico, as que estiverem a 4 espaços da margem esquerda fazem parte do segundo nível, aquelas que estiverem a 8 espaços fazem parte do terceiro nível e assim por diante.

Todos os blocos são delimitados pela profundidade da indentação. O mau uso do recurso da indentação, por exemplo, não utilizar indentação onde é necessária ou utilizar 4 espaçamentos onde o correto seria 8 espaçamentos, poderá resultar no mau funcionamento ou na não execução do programa.



Exemplo 1 - Uso correto da indentação:

```
y = 10                # primeiro nível hierárquico
if y%2 == 0:          # primeiro nível hierárquico
    print("y é par")   # segundo nível hierárquico
else:
    print("y é ímpar") # segundo nível hierárquico
```

Resultado:

```
y é par
```

Exemplo 2 - Indentação não utilizada onde é necessária:

```
y = 10
if y % 2 == 0:
print('y é par')
else:
print('y é ímpar')
```

Resultado:

```
SyntaxError: invalid syntax
```

```
>>> print('y é ímpar')
```

```
y é ímpar
```

2. Ambiente de desenvolvimento

Objetivo: Apresentar o ambiente básico de desenvolvimento do Python conhecido como IDLE.

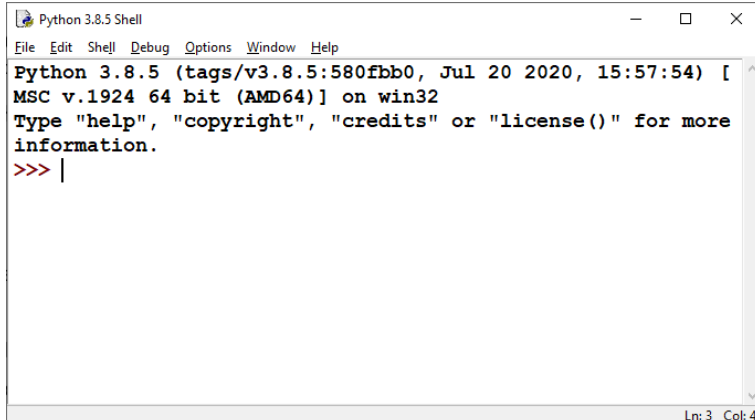
É possível utilizar diferentes IDE's (Integrated Development Environment) para programar em Python. A ferramenta mais simples é conhecida pela sigla IDLE.

IDLE

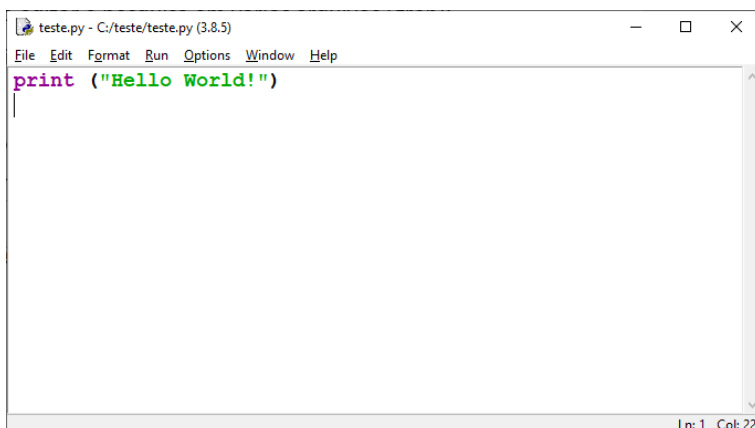
IDLE (Integrated Development and Learning Environment) é o ambiente integrado de desenvolvimento e aprendizagem do Python. O IDLE possui os seguintes recursos:

- Codificado em Python 100% puro, usando o kit de ferramentas tkinter GUI;
- Multiplataforma: funciona basicamente da mesma forma no Windows, Unix e macOS;
- Janela de shell Python (interpretador interativo) com colorização de entrada de código, saída e mensagens de erro;
- Editor de texto de várias janelas com desfazer múltiplo, colorização Python, recuo inteligente, dicas de chamada, preenchimento automático e outros recursos;
- Pesquisa em qualquer janela, substituição nas janelas do editor e pesquisa em vários arquivos (grep);
- Depurador com pontos de interrupção persistentes, revisão e visualização de namespaces;
- etc.

O IDLE possui dois tipos de janela principal, a janela **Shell** (interpretador) e a janela **Editor**. É possível ter várias janelas do Editor simultaneamente. No Windows e no Linux, cada uma tem seu próprio menu superior.



IDLE - Shell (interpretador)



IDLE - Editor

Primeiro programa em Python

1º Digitar o seguinte na janela Editor:

```
print("Hello World!")
```

A função **print** é usada para imprimir o conteúdo na tela.


O Python é case sensitive, isto é, diferencia maiúsculas de minúsculas. Portanto print é diferente de Print.

2º Salvar o programa acima com o nome **teste.py**.

Os programas desenvolvidos em Python têm por padrão a extensão **.py**.

3º Selecionar a opção **Run** na janela Editor e clicar em **Run Module**. Outra alternativa é simplesmente pressionar a tecla F5.

Ao concluir estes passos a o seguinte resultado será exibido na janela Shell (interpretador):



```
Python 3.8.5 Shell
File Edit Shell Debug Options Window Help
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [
MSC v.1924 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
===== RESTART: C:/teste/teste.py =====
Hello World!
>>> |
```

IDLE - Shell - Saída de teste.py

Usando o Shell (interpretador) como calculadora

O interpretador Python pode ser utilizado como uma calculadora, conforme observado a seguir:



```
Python 3.8.5 Shell
File Edit Shell Debug Options Window Help
>>>
>>> 10+4
14
>>> 10-4
6
>>> 10*4
40
>>> 10/4
2.5
>>> 10**4
10000
>>> 10%4
2
>>>
```

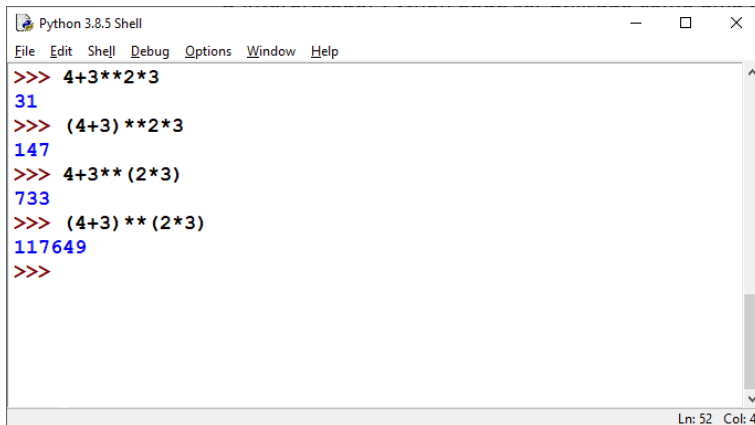
IDLE - Shell - Calculadora

Os parênteses são aplicados pelo Python da mesma forma que em expressões matemáticas, isto é, com as seguintes prioridades:

1ª Exponenciação **

2ª Multiplicação e divisão * e /

3ª Adição e subtração + e -

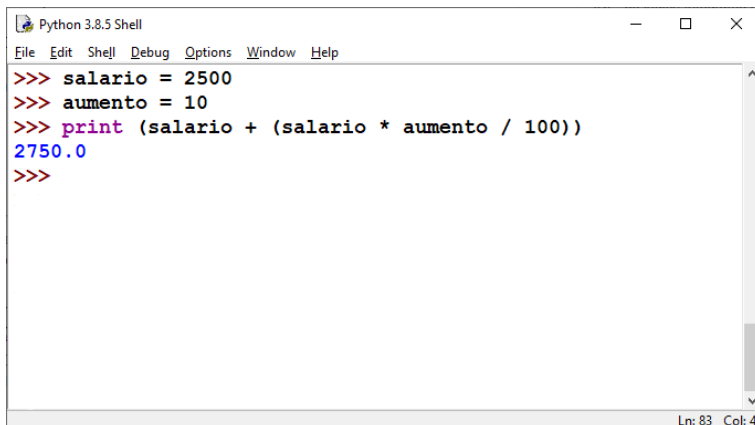


```
Python 3.8.5 Shell
File Edit Shell Debug Options Window Help
>>> 4+3**2*3
31
>>> (4+3)**2*3
147
>>> 4+3*(2*3)
733
>>> (4+3)**(2*3)
117649
>>>
```

IDLE - Shell - Expressões matemáticas

Trabalhando com variáveis no Shell

O Shell (interpretador) também pode efetuar operações mais complexas utilizando variáveis, conforme segue:



```
Python 3.8.5 Shell
File Edit Shell Debug Options Window Help
>>> salario = 2500
>>> aumento = 10
>>> print(salario + (salario * aumento / 100))
2750.0
>>>
```

IDLE - Shell - Operações com variáveis

Entrada de dados

A função **input** é usada para solicitar a entrada de dados. A função recebe um parâmetro, isto é, a mensagem a ser exibida, e retorna o valor digitado pelo usuário.

O código a seguir deverá ser digitado no Editor do IDLE:

```
x = input("Digite um número: ")
print(x)
```

O código deverá ser salvo com a extensão **.py** (**teste.py**, por exemplo)

A seguir, na opção **Run** do menu superior, deve-se selecionar a opção Run Module.

Será exibida na tela do Shell do IDLE será exibida a mensagem:

Digite um número:

Quando um número for digitado (12, por exemplo) a tela do Shell exibirá o valor digitado:

12



A função **input()** sempre retorna uma **string**. A conversão do valor retornado em **int** é realizada com a função **int()** e a conversão do valor retornado em **float** é realizada com a função **float()**, conforme apresentado no exemplo a seguir:

```
idade = int(input("Idade: "))  
print(f"Idade: {idade}")  
salario = float(input("Salário: "))  
print(f"Salário: {salario:5.2f}")
```

3. Variáveis, tipos de dados e operadores

Objetivo: Apresentar a criação e a atribuição de valores através de variáveis, os principais tipos de dados e os operadores da linguagem Python.

Variáveis

Variáveis, no contexto de programação, são utilizadas para armazenar valores e para dar nomes a áreas da memória do dispositivo (computador, por exemplo) onde os dados são armazenados.

A atribuição de valores às variáveis no Python, como em muitas outras linguagens é realizada através do operador de atribuição `=`.

Os exemplos a seguir demonstram como atribuir valores às variáveis denominadas **x** e **y**:

```
x = 5 # x recebe 5
y = "teste" # y recebe "teste"
```

É possível também atribuir o valor armazenado em uma variável para outra variável:

```
a = 10
b = a
```

No exemplo apresentado, o valor armazenado na variável **a** foi copiado para variável **b**. Portanto, as duas variáveis armazenam agora o mesmo valor: **10**.

É importante enfatizar que, a partir deste momento, alterar o valor da variável **a** não terá como consequência alterar o valor da variável **b** e vice e versa. Cada variável utiliza um endereço de memória diferente, portanto podem armazenar no mesmo momento, se necessário, valores diferentes.

```
a = 10
b = a
a = 20 # a passa a valer 20 e b continua valendo 10
```

Nomes de variáveis

Na linguagem Python, nomes de variáveis devem obrigatoriamente começar com uma letra ou com `_` (underscore). A partir do segundo caractere, números também podem ser usados para nominar variáveis. Nomes de variáveis não podem conter espaços em branco. A versão 3 utiliza o conjunto de caracteres UTF-8 e, portanto, permite a utilização de acentos em nomes de variáveis.

Tipos de dados

Em Python tudo é objeto. Isso quer dizer que um objeto do tipo **string**, por exemplo, tem seus próprios métodos.

O conceito de variável é uma associação entre um **nome** e um **valor**, mas não é necessário declarar o tipo da variável, portanto, o tipo relacionado a variável pode variar durante a execução do programa.

Tipos numéricos:

- inteiro (int)
- ponto flutuante (float)
- complexo (complex)

Os exemplos apresentados a seguir usam a função **type()** que retorna o tipo de cada variável.

Inteiro:

```
>>> x = 10
>>> type(x)
<type 'int'>
```

Ponto flutuante:

```
>>> x = 10.0
>>> type(x)
<type 'float'>
```

Complexo:

```
>>> x = 2+3j
>>> type(x)
<type 'complex'>
```

Tipos booleanos

Tipos booleanos armazenam apenas dois valores lógicos **True** (verdadeiro) ou **False** (falso). Devem obrigatoriamente ser escritos **True** e **False** (com o primeiro caractere em maiúscula).

- booleano (bool)

```
>>> x = True
>>> type(x)
<type 'bool'>
```

Strings

Strings são usadas para armazenar uma cadeia (conjunto) de caracteres:

- string (str)

```
>>> x = "teste"
>>> type(x)
<type 'str'>
```

Operadores

O Python disponibiliza os seguintes tipos de operadores:

- relacionais
- lógicos
- aritméticos

Operadores relacionais

São utilizados para realizar comparações lógicas. O Python suporta os operadores relacionais apresentados a seguir:

Operador	Operação	Descrição
==	igual	Retorna True se o primeiro operando for igual ao segundo, senão retorna False.
<	menor	Retorna True se o primeiro operando for menor que segundo, senão retorna False.
<=	menor ou igual	Retorna True se o primeiro operando for menor ou igual ao segundo, senão retorna False.
>	maior	Retorna True se o primeiro operando for maior que segundo, senão retorna False.
>=	maior ou igual	Retorna True se o primeiro operando for maior ou igual ao segundo, caso contrário retorna False.
!=	diferente	Retorna True se o primeiro operando for diferente do segundo, caso contrário retorna False.

Operadores aritméticos

São utilizados para realizar operações matemáticas. O Python suporta os operadores aritméticos apresentados a seguir:

Operador	Operação	Descrição
+	adição	Retorna a soma de duas variáveis ou expressões.
-	subtração	Retorna a diferença entre duas variáveis ou expressões.
*	multiplicação	Retorna a multiplicação de duas variáveis ou expressões.
/	divisão	Retorna a divisão de duas variáveis ou expressões.
**	exponenciação	Retorna o valor da primeira variável elevada a segunda variável.
//	parte inteira da divisão	Retorna a parte inteira da divisão da primeira variável pela segunda.
%	módulo da divisão	Retorna o resto da divisão da primeira variável pela segunda.

Operadores de atribuição especiais

São utilizados para simplificar a escrita. O Python suporta os operadores de atribuição especiais apresentados a seguir:

Operador	Exemplo	Equivalência
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
**=	a **= b	a = a ** b
//=	a //= b	a = a // b

Operadores lógicos

São utilizados para agrupar operações com lógica booleana. O Python suporta os operadores lógicos apresentados a seguir:

Operador	Operação	Descrição
and	conjunção	Retorna True apenas quando as duas expressões retornarem True.
or	disjunção	Retorna True quando pelo menos uma das expressões retornar True.
not	negação	Um valor True negado retorna False e vice-versa.

Quando uma expressão apresentar mais de um operador lógico, o operador **not** é avaliado primeiro, seguido do operador **and** e, finalmente o operador **or**.

4. Strings

Objetivo: Apresentar as operações com strings, como utilizar os marcadores de posição e como realizar a entrada de dados durante a execução dos programas.

Strings em Python (como muitas outras linguagens de programação) são arrays de bytes que representam caracteres Unicode. No entanto, o Python não tem um tipo de dados de caractere, um único caractere é simplesmente uma string com comprimento 1 (um). Os colchetes podem ser usados para acessar os elementos da string.

Operações com strings

As strings suportam as seguintes operações:

- concatenação
- composição
- fatiamento
- **Concatenação**

Concatenação é a junção de duas ou mais strings em uma nova string maior. Para concatenar duas strings é usado o operador + (mais), conforme demonstrado a seguir:

```
>>> x = "ABC"
>>> y = "DEF"
>>> x + y
'ABCDEF'
```

Utilizando o operador * (multiplicação) é possível repetir uma string várias vezes, conforme os seguintes exemplos:

```
>>> "a" * 5
'aaaaa'

>>> "a" + "b" * 3
'abbb'
```

Porém, a tentativa de juntar variáveis de diferentes tipos para formar uma string resultará em erro, conforme pode ser observado a seguir:

```
>>> nome = "John"
>>> idade = 35
>>> nome + idade
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

- **Composição**

A junção de strings que apresentam tipos diferentes pode ser feita através da composição, conforme demonstrado a seguir:

```
>>> "%s %d" % ("John", 35)
'John 35'
```

O **%s** e o **%d** observados no exemplo acima são chamados de marcadores de posição. O **%s** indica que naquela posição será inserido uma string e o **%d** indica que será inserido um inteiro. Os principais tipos de marcadores são apresentados a seguir:

- **%d** - usado para números inteiros
- **%f** - usado para números decimais (float)
- **%s** - usado para strings

A formatação de número decimais pode ser realizada com dois valores entre o **%** e o **f**. O primeiro valor indica a quantidade total de caracteres a reservar; e o segundo, o número de casas decimais.

```
>>> "%5f" % 3
'3.000000'
```

```
>>> "%5.2f" % 3
' 3.00'
```

A composição de string também pode ser realizada utilizando a função **format()**:

```
nome = "John"
idade = 35
salario = 5000.00

("{} tem {} anos e seu salário é {}".format(nome, idade, salario))
'John tem 35 anos e seu salário é 5000.0.'
```

```
("{:10} tem {:3} anos e seu salário é {:.5.2f}".format(nome, idade, salario))
'John          tem 35 anos e seu salário é 5000.00.'
```

Há uma terceira forma de compor strings com o uso de **f-strings**:

```
f"{nome} tem {idade} anos e seu salário é {salario:5.2f}"
'John tem 35 anos e seu salário é 5000.00'
```

- **Fatiamento (Slicing)**

O fatiamento retorna um intervalo de caracteres a partir do índice inicial (inclusive) até o índice final (não incluído). Os dois índices devem ser separados por dois pontos, para retornar uma parte da string. O índice inicial de uma string é igual a 0 (zero), o segundo é igual a 1 (um) e assim por diante. Portanto, uma string com 5 (cinco) caracteres, por exemplo, terá o índice inicial 0 (zero) e o final 4 (quatro).

Os exemplos a seguir apresentam alguns fatiamentos realizados na string **ABCDEFGH**:

```
>>> x = "ABCDEFGH"

>>> x[0:3]
'ABC'

>>> x[2:4]
'CD'

>>> x[:2]
'AB'

>>> x[2:]
'CDEFGH'
```

```
>>> x[:]  
'ABCDEFGH'  
  
>>> x[-1]  
'H'  
  
>>> x[-4:7]  
'EFG'  
  
>>> x[-2:-1]  
'G'
```

Verificação parcial

A função **startswith** verifica se uma string começa com determinados caracteres e a função **endswith** verifica se a string termina com determinados caracteres.

O código a seguir deverá ser digitado no Editor do IDLE:

```
>>> nome = "John Smith"  
>>> nome.startswith("John")  
True  
  
>>> nome = "John Smith"  
>>> nome.endswith("John")  
False  
  
>>> nome = "John Smith"  
>>> nome.endswith("Smith")  
True
```

A função **startswith** e **endswith** diferenciam letras maiúsculas e minúsculas. Portanto, em determinadas ocasiões será necessário converter a string para maiúsculas usando a função **upper** ou para minúsculas usando a função **lower**.

```
>>> nome = "John Smith"  
>>> nome.upper()  
'JOHN SMITH'  
  
>>> nome.lower()  
'john smith'  
  
>>> nome.upper().startswith('JOHN')  
True  
  
>>> nome.lower().startswith('john')  
True
```

O operador **in** também pode ser usado para verificar se um ou mais caracteres fazem parte da string.

```
>>> nome = "John Smith"  
>>> "Smith" in nome  
True  
  
>>> "smith" in nome  
False  
  
>>> "J" in nome  
True
```

Contagem de ocorrências

A função **count** retorna a quantidade de um caractere ou palavra em uma string.

O código a seguir deverá ser digitado no Editor do IDLE:

```
>>> s = "John Smith"
>>> s.count("h")
2
>>> s.count("John")
1
```

Pesquisa

A função **find** verifica se uma string é parte de outra e devolve a posição que corresponde a primeira ocorrência. Se a string pesquisada não for encontrada a função retornará -1.

```
>>> s = "John Smith"
>>> s.find("John")
0
>>> s.find("Smith")
5
>>> s.find("h")
2
>>> s.find("Mary")
-1
```

A função **rfind** verifica se uma string é parte de outra, porém realiza a pesquisa da direita para a esquerda.

```
>>> s = "John Smith"
>>> s.rfind("h")
9
```

As duas funções, **find** e **rfind**, também aceitam dois parâmetros adicionais que correspondem, respectivamente, às posições de início e fim a considerar na pesquisa.

```
>>> s = "Knowledge is power"
>>> s.find("w")
3
>>> s.find("w", 5) # início = 5
15
>>> s.find("w", 5, 10) # início = 5 e fim = 10
-1
```

As funções **find** e **rfind** podem ser usadas para verificar todas as ocorrências de um determinado caractere ou de uma string dentro de outra string.

```
s = "Knowledge is power"
n = 0
while(n > -1):
    n = s.find("w", n)
    if n >= 0:
        print(f"Encontrado na posição: {n}")
        n += 1
```

```
Encontrado na posição: 3
Encontrado na posição: 15
```

Substituição de strings

A função **replace** substitui caracteres ou trechos de uma string. A função recebe dois parâmetros, o primeiro corresponde à string que será substituída e o segundo corresponde à string que a substituirá. Caso seja passado uma string vazia como segundo parâmetro, o trecho será excluído da string.

```
>>> s = "John Smith"
>>> s.replace("Smith", "Smart")
'John Smart'

>>> s = "John Smith"
>>> s.replace("Smith", "")
'John '
```

Remoção de espaços

A função **strip** é usada para remover espaços em branco do início ou fim da string. Outras duas funções **lstrip** e **rstrip**, removem espaços em branco, respectivamente, dos lados **esquerdo** e **direito** da string.

```
>>> s = "  John Smith  "
>>> s.strip()
'John Smith'

>>> s = "  John Smith  "
>>> s.lstrip()
'John Smith  '

>>> s = "  John Smith  "
>>> s.rstrip()
'  John Smith'
```

Separação de strings

A função **split** separa uma string a partir de um caractere apresentado como parâmetro e retorna uma lista com substrings separadas.

```
>>> s = "John Smith, Mary Clark, Noah Baker"
>>> s.split(",")
['John Smith', ' Mary Clark', ' Noah Baker']
```

5. Estruturas de decisão

Objetivo: Apresentar as estruturas de decisão if, if-elif e if-elif-else, conforme a sintaxe da linguagem Python.

if

A estrutura **if** verifica a condição a cada passagem. Se a condição for verdadeira (True), então a lista de instruções a seguir será executada. Se a condição for falsa (False), então serão executadas as próximas instruções.

O exemplo apresentado a seguir solicita que sejam informados dois valores. A primeira condição avaliada é se o valor atribuído à variável **a** é menor que o valor atribuído à variável **b**. Se essa condição for verdadeira, imprimirá "O primeiro valor é menor". Nessa estrutura, a segunda condição, independentemente da primeira ser verdadeira, também será avaliada. A segunda condição avaliará se o valor atribuído a variável **b** é menor que o atribuído a variável **a** e, se a condição for verdadeira, imprimirá "O segundo valor é menor".

```
a = int(input("Primeiro valor: "))
b = int(input("Segundo valor: "))
if a < b:
    print("O primeiro valor é menor.")
if b < a:
    print("O segundo valor é menor.")
```

if-elif

Na estrutura **if-elif** as expressões de teste são avaliadas de cima para baixo. Assim que uma expressão verdadeira é encontrada, o comando associado a ela é executado.

O exemplo apresentado a seguir solicita que seja informada a nota do aluno. Se a nota for menor que 5 o aluno estará "Reprovado"; se a nota for menor que 7 o aluno deverá fazer o "Exame" e; se a nota for igual ou maior que 7 o aluno estará "Aprovado". Nessa estrutura, se a primeira condição for verdadeira, as demais não serão avaliadas. O mesmo ocorrerá se a segunda condição for verdadeira, a terceira condição não será avaliada.

```
nota = float(input("Informe a nota do aluno: "))
if nota < 5:
    print("Reprovado.")
elif nota < 7:
    print("Exame")
elif nota >= 7:
    print("Aprovado")
```

if-elif-else

Na estrutura **if-elif-else** o comando **else** executará uma instrução (ou instruções) se as expressões de teste dos comandos **if** e **elif** forem falsas.

```
a = int(input("Primeiro valor: "))
b = int(input("Segundo valor: "))
if a < b:
    print("O primeiro valor é menor.")
elif b < a:
    print("O segundo valor é menor.")
else:
    print("Os valores são iguais.")
```



Estruturas aninhadas

Muitas vezes para obter o resultado desejado será necessário construir estruturas aninhadas, posicionando um **if** "dentro" de outro.

O exemplo a seguir apresenta os valores pagos de acordo com a quantidade adquirida de determinado produto. Conforme a quantidade adquirida o cliente pagará **valores unitários diferenciados** ao fornecedor. Para quantidade inferior a 100 peças o valor unitário corresponderá a 1.80; para quantidade inferior a 200 peças o valor unitário corresponderá a 1.50; e para quantidade superior o valor unitário corresponderá a 1.20. A seguir, o programa multiplicará a quantidade pelo valor unitário obtido a partir da estrutura de decisão.

```
quant = float(input("Informe quantidade: "))
if quant < 100:
    valor = 1.80
else:
    if quant < 200:
        valor = 1.50
    else:
        valor = 1.20
print(f"Valor total: {quant * valor:6.2f}")
```


6. Estruturas de repetição

Objetivo: Apresentar a estruturas de repetição while, simples e aninhadas, conforme a sintaxe da linguagem Python.

As estruturas de repetição são usadas para executar a mesma parte de um programa mais de uma vez e, frequentemente, dependendo da condição.

while

A estrutura de repetição criada com o comando **while** repete um bloco de instruções enquanto a condição for verdadeira.

O programa apresentado a seguir imprimirá o valor atribuído a x, incrementado a cada iteração, 10 vezes.

```
x = 1
while x <= 10:
    print(x)
    x = x + 1
```

Contadores

Um contador é utilizado para contar o número de vezes que uma instrução ocorre, ou seja, contar a quantidade de vezes que uma instrução é executada.

O exemplo a seguir apresenta um programa no qual o último número a ser impresso será informado pelo usuário.

```
final = int(input("Número de vezes a imprimir: "))
x = 1
while x <= final:
    print(x)
    x = x + 1
```

Acumuladores

No desenvolvimento de programas, além de contadores, muitas vezes é necessário também usar acumuladores. A diferença entre contadores e acumuladores é que nos contadores os valores adicionados à variável são constantes e nos acumuladores os valores sofrem variações, conforme a instrução que está sendo executada.

O exemplo a seguir apresenta um programa que realizará a soma de 5 valores que serão informados pelo usuário. No exemplo, observa-se que o valor adicionado à variável n é constante, portanto **n** é um **contador**. Por outro lado, os valores atribuídos à variável soma sofrem variações a cada iteração, portanto **soma** é um **acumulador**.

```
n = 1
soma = 0
while n <= 5:
    x = int(input(f"Número {n}:"))
    soma = soma + x
    n = n + 1
print(f"Soma: {soma}")
```

break

O comando **break** é utilizado para interromper a execução de uma estrutura de repetição (**while**) independente da condição ser verdadeira.

No exemplo apresentado a seguir, a condição de **while** foi substituída por True. A estrutura de repetição, sob esta condição, será executada para sempre, pois sua condição de parada (**True**) é constante. O comando **break**, executado quando o usuário digitar 0 (zero), interrompe a execução do **while**, independente da sua condição.

```
soma = 0
while True:
    x = int(input("Número a somar ou 0 para sair: "))
    if x == 0:
        break
    soma += x
print(soma)
```

A instrução **soma += x** equivale a **soma = soma + x** (veja: *Operadores de atribuição especiais em Variáveis, tipos de dados e operadores*)

Repetições aninhadas

A linguagem Python, assim como a maioria das linguagens de programação, permite o alinhamento de várias estruturas **while**. Neste tipo de construção é essencial observar a indentação correta para não obter resultados indesejáveis, visto que Python não { } (chaves) ou a palavra reservada **begin** para delimitar blocos de instruções.

O programa a seguir imprimirá as tabuadas de 1 a 10, utilizando o recurso de repetições aninhadas.

```
x = 1
while x <= 10:
    y = 1
    while y <= 10:
        print(f"{x} x {y} = {x * y}")
        y += 1
    x += 1
```

7. Listas

Objetivo: Apresentar as características e as principais operações realizadas em listas na linguagem Python.

Listas são um tipo de variável que permitem o armazenamento de 0 (zero) ou mais elementos de um mesmo tipo ou de tipos diferentes. Cada valor armazenado em uma lista é acessado por um índice. O tamanho da lista é igual a quantidade de elementos que ela contém.

O exemplo apresentado a seguir cria uma lista com três elementos. Após o identificador (L) e o sinal de igualdade, utiliza-se [] (colchetes) para delimitar os elementos da lista.

```
L = [3, 5, 7]
```

O primeiro elemento tem índice 0 (zero), o segundo tem índice 1 (um) e assim por diante.

```
>>> L = [3,5,7]
>>> L[0]
3
```

Para alterar o valor do primeiro elemento da lista é necessário apenas referenciá-lo usando o seu índice e, em seguida, atribuir-lhe o novo valor.

```
>>> L[0] = 9
>>> L[0]
9
```

O exemplo a seguir apresenta o cálculo da média de um aluno com base em uma lista com quatro elementos.

```
nota = [3, 5, 7, 9]
soma = 0
x = 0
while x < 4:
    soma += nota[x]
    x += 1
print(f"Media: {soma / x:5.2f}")
```

O exemplo a seguir, uma variação do anterior, apresenta o cálculo da média de um aluno com base em uma lista com quatro elementos inseridos pelo usuário.

```
nota = [0, 0, 0, 0]
soma = 0
x = 0
while x < 4:
    nota[x] = float(input(f"Nota {x}: "))
    soma += nota[x]
    x += 1
print(f"Media: {soma / x:5.2f}")
```

Copiando uma lista

Listas em Python são objetos. Portanto, são necessários alguns cuidados para não se obter resultados indesejados ao fazer cópias de listas.

O exemplo abaixo apresenta o que **aparentemente** seria uma cópia independente de uma lista.

```
>>> L = [3, 5, 7, 9]
```

```
>>> L
```

```
[3, 5, 7, 9]
```

```
>>> Z = L
```

```
>>> Z
```

```
[3, 5, 7, 9]
```

A seguir o elemento de índice 0 (zero) da lista Z é alterado para 2:

```
>>> Z[0] = 2
```

```
>>> Z
```

```
[2, 5, 7, 9]
```

```
>>> L
```

```
[2, 5, 7, 9]
```

No exemplo apresentado foi copiado apenas uma referência da lista e não os seus dados. Z é apenas um apelido para L, portanto, Z e L são a mesma lista.

A criação de uma cópia independente de uma lista deve ser realizada conforme segue:

```
>>> L = [3, 5, 7, 9]
```

```
>>> L
```

```
[3, 5, 7, 9]
```

```
>>> Z = L[:] # Cópia independente
```

```
>>> Z
```

```
[3, 5, 7, 9]
```

```
>>> Z[0] = 2
```

```
>>> Z
```

```
[2, 5, 7, 9]
```

```
>>> L
```

```
[3, 5, 7, 9]
```

Fatiando uma lista

O fatiamento de listas é muito similar ao fatiamento de strings, conforme exemplos apresentados a seguir.

```
>>> L
```

```
[3, 5, 7, 9]
```

```
>>> L[0:4]
```

```
[3, 5, 7, 9]
```

```
>>> L[:4]
```

```
[3, 5, 7, 9]
```

```
>>> L[:-1]
```

```
[3, 5, 7]
```

```
>>> L[1:3]
```

```
[5, 7]
```

```
>>> L[2:]
```

```
[7, 9]
```

```
>>> L[-1]
```

```
9
```

Tamanho de listas

A função **len** retorna o número de elementos de uma lista.

```
>>> L = [3, 5, 7, 9]
>>> len(L)
```

4

A função **len** também pode ser usado para controlar o limite dos índices. O exemplo apresentado a seguir percorre todos os elementos de uma lista usando uma estrutura de repetição.

```
L = [3, 5, 7, 9]
x = 0
while x < len(L):
    print(L[x])
    x += 1
```

3

5

7

9

Adição de elementos

A função **append** adiciona um elemento no final da lista.

```
>>> L = ["a", "b"]
>>> L
['a', 'b']
>>> L.append("c")
>>> L
['a', 'b', 'c']
```

A adição de novos elementos à lista também pode ser realizada com o operador **+**.

```
>>> L
['a', 'b', 'c']
>>> L = L + ["d"]
>>> L
['a', 'b', 'c', 'd']
```

O programa a seguir adiciona elementos (números inteiros) à lista até que 0 (zero) seja pressionado.

```
L = []
while True:
    n = int(input("Digite um número inteiro ou 0 para sair: "))
    if n == 0:
        break
    L.append(n)
x = 0
while x < len(L):
    print(L[x])
    x += 1
```

Remoção de elementos

A instrução **del** para remove elementos de uma lista.

```
>>> L = ["a", "b", "c"]
>>> L
['a', 'b', 'c']
>>> del L[2]
>>> L
['a', 'b']
```

É possível remover fatias inteiras de uma lista. O exemplo a seguir cria uma lista com números inteiro de 1 a 10. (A função **range** será apresentada em detalhes nos próximos tópicos.)

```
>>> L = list(range(11))
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
del L[0:5]
L
[5, 6, 7, 8, 9, 10]
```

Pesquisa

É possível realizar uma pesquisa sequencial em uma lista utilizando estruturas de repetição, conforme apresentado a seguir.

```
L = [1, 3, 5, 7]
n = int(input("Valor a procurar: "))
localizou = False
x = 0
while x < len(L):
    if L[x] == n:
        localizou = True
        break
    x += 1
if localizou:
    print(f"{n} encontrado na posição {x}")
else:
    print(f"{n} não encontrado")
```

A instrução **for** funciona de forma parecida com **while**, porém a cada repetição utiliza um elemento diferente da lista.

```
L = [1, 3, 5, 7]
for n in L:
    print(n)
```

A pesquisa sequencial também pode ser construída com a instrução **for** conforme apresentado a seguir.

```
L = [1, 3, 5, 7]
n = int(input("Valor a procurar: "))
for x in L:
    if x == n:
        print("Valor encontrado.")
        break
else:
    print("Valor não encontrado")
```

Range

A função **range** é um gerador (generator) de valores. O exemplo a seguir gera números sequenciais de 0 a 4.

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

```
0  
1  
2  
3  
4
```

A função **range** permite também definir a valor inicial da sequência. O exemplo a seguir gera números sequenciais de 5 a 9.

```
for n in range(5, 10):  
    print(n)
```

```
5  
6  
7  
8  
9
```

A função **range** aceita um terceiro parâmetro que determina o valor de "saltos" entre os elementos gerados. O exemplo a seguir gera números sequenciais de iniciando com 0 (zero) e com saltos de 2 em 2.

```
for n in range(0, 10, 2):  
    print(n)
```

```
0  
2  
4  
6  
8
```

Os valores gerados com a função **range** não formam exatamente uma lista. Para transformar um gerador em lista é preciso fazer uso da função **list** como demonstrado a seguir.

```
L = list(range(0, 10, 2))  
print(L)  
[0, 2, 4, 6, 8]
```

Enumerate

A função **enumerate** amplia as funcionalidades da instrução **for**. O exemplo a seguir imprime uma lista com os índices entre [] (colchetes) e os valores à direita dos respectivos índices.

```
L = [1, 3, 5, 7]  
for x, n in enumerate(L):  
    print(f"[{x}] {n}")
```

```
[0] 1  
[1] 3  
[2] 5  
[3] 7
```

Ordenação

O Python apresenta duas funções - **sort** e **sorted** - para realizar a ordenação de elementos de listas.

sort

A função **sort** altera a posição dos elementos da lista para realizar a ordenação.

```
>>> L = [5, 3, 7, 1]
>>> L.sort()
>>> L
[1, 3, 5, 7]
```

sorted

A função **sorted** apresenta os valores da lista ordenados, mantendo a lista inalterada.

```
>>> L = [5, 3, 7, 1]
>>> sorted(L)
[1, 3, 5, 7]
>>> L
[5, 3, 7, 1]
```

Empacotamento e desempacotamento

Empacotar elementos em uma lista envolve atribuir seus elementos individuais a uma lista. Desempacotar uma lista, é exatamente o contrário, envolve atribuir os elementos dela individualmente a outros objetos. O empacotamento e desempacotamento de uma lista é apresentado a seguir:

```
>>> L = [1, 3, 5]
>>> x, y, z = L
>>> x
1
>>> y
3
>>> z
5
>>> L = [1, 3, 5]
>>> L
```

A quantidade de variáveis usadas para realizar o desempacotamento sempre deve ser igual a quantidade de elementos da lista.

O caractere ***** é usado para representar vários elementos a desempacotar:

```
>>> *x, y = [1, 3, 5, 7]
>>> x
[1, 3, 5]
>>> y
7
>>> x, *y, z = [1, 3, 5, 7]
>>> x
1
>>> y
[3, 5]
>>> z
7
```


Soma, máximo e mínimo de uma lista

O Python disponibiliza as funções `sum()`, `max()` e `min()` para calcular respectivamente a soma, o valor máximo e o valor mínimo de uma lista.

```
>>> L = [1, 3, 5, 7]
>>> sum(L)
16
>>> max(L)
7
>>> min(L)
1
```

List comprehension

List Comprehension é uma forma simples e concisa de criar listas. O exemplo apresentado a seguir cria uma lista com os números inteiros de 0 a 9.

```
>>> L = [x for x in range(10)]
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

O exemplo seguinte cria uma lista com os números pares de 0 a 9.

```
>>> L = [x for x in range(10) if x % 2 == 0]
>>> L
[0, 2, 4, 6, 8]
```

O próximo exemplo cria uma lista com os caracteres A, B, C e D.

```
>>> L = [x.upper() for x in "abcd"]
>>> L
['A', 'B', 'C', 'D']
```

8. Dicionários

Objetivo: Apresentar as características e operações com estruturas de dados denominadas dicionários na linguagem Python.

Dicionários apresentam uma estrutura de dados similar às listas, mas têm propriedades de acesso diferentes. Cada dicionário é composto por um conjunto de chaves e valores. Cada elemento do dicionário é uma combinação de chave e valor. Enquanto nas listas cada índice é um número inteiro, os dicionários utilizam as chaves como índice.

A partir da tabela apresentada a seguir será criado um dicionário denominado **estado**.

estado	
sigla	nome
AM	Amazonas
SP	São Paulo
PR	Paraná

```
>>> estado = {"AM": "Amazonas",  
              "SP": "São Paulo",  
              "PR": "Paraná"}
```

Para obter, por exemplo, o nome do estado com chave igual a "SP" é preciso informar o nome da variável tipo dicionário (**estado**) e a chave entre colchetes ["SP"].

```
>>> estado["SP"]  
São Paulo
```

O operador **in** é utilizado para verificar se uma chave pertence a um determinado dicionário.

```
>>> print("RS" in estado)  
False  
>>> print("SP" in estado)  
True
```

A função **keys** retorna as chaves do dicionário.

```
>>> print(estado.keys())  
dict_keys(['AM', 'SP', 'PR'])
```

A função **values** retorna as chaves do dicionário.

```
>>> print(estado.values())  
dict_values(['Amazonas', 'São Paulo', 'Paraná'])
```

A instrução **del** apaga uma chave (e seu respectivo valor) de um dicionário.

```
>>> del estado["PR"]  
>>> print(estado)  
{'AM': 'Amazonas', 'SP': 'São Paulo'}
```

Dicionários com listas

Python oferece o recurso de criação de dicionários nos quais as chaves são associadas a listas ou a outros dicionários.



estado		
sigla	nome	região
AM	Amazonas	Norte
SP	São Paulo	Sudeste
PR	Paraná	Sul

```
>>> estado = {"AM": ["Amazonas", "Norte"],
               "SP": ["São Paulo", "Sudeste"],
               "PR": ["Paraná", "Sul"]}

>>> estado["SP"]
['São Paulo', 'Sudeste']
```

9. Tuplas

Objetivo: Apresentar as características e principais operações com tuplas na linguagem Python.

Tuplas, em Python, são similares as listas, porém são **imutáveis**, isto é, não podem ter seus elementos alterados. Devem ser utilizados () parênteses no lugar de [] colchetes para criação das tuplas.

```
>>> T = ("a", "b", "c", "d")
>>> T
('a', 'b', 'c', 'd')
```

Nota: Tuplas podem ser criadas sem a utilização dos () parênteses, porém o uso destes torna o código mais legível.

As tuplas suportam a maior parte das operações realizadas em listas, conforme os seguintes exemplos.

Fatiamento

```
>>> T[0:2]
('a', 'b')
```

Tamanho

```
>>> len(T)
4
```

Criação de tuplas a partir de listas

Tuplas também podem ser criadas a partir de listas, conforme o exemplo apresentado a seguir.

```
>>> L = [1, 3, 5, 7]
>>> T = tuple(L)
>>> T
(1, 3, 5, 7)
```

Concatenação

Embora não seja possível alterar os elementos de uma tupla, a operação de concatenação de tuplas é possível, conforme apresentado a seguir.

```
>>> T1 = (1, 2, 3)
>>> T2 = (4, 5, 6)
>>> T1 + T2
(1, 2, 3, 4, 5, 6)
```

Alteração de objetos internos à tupla

Listas ou outros objetos que podem ser alterados, contidos em tuplas, continuarão funcionando normalmente, isto é, poderão ser alterados mesmo que componham os elementos da tupla. No exemplo a seguir a lista ["c", "d"] é o terceiro elemento da tupla e pode ser alterado normalmente.

```
>>> T = ("a", "b", ["c", "d"])
>>> T
('a', 'b', ['c', 'd'])
>>> len(T)
3
>>> T[2]
['c', 'd']
>>> T[2].append("e")
>>> T
('a', 'b', ['c', 'd', 'e'])
```

Empacotamento e desempacotamento

As operações de empacotamento e desempacotamento, já explicadas em **listas** também funcionam para tuplas. A instrução **t = 1, 3, 5**, apresentada a seguir, é um exemplo de empacotamento de tupla: 1, 3 e 5 são empacotados em uma tupla.

```
>>> t = 1, 3, 5
>>> t
(1, 3, 5)
```

A operação inversa, o desempacotamento, também é possível:

```
>>> x, y, z = t
>>> x
1
>>> y
3
>>> z
5
```

A quantidade de variáveis usadas para realizar o desempacotamento sempre deve ser igual a quantidade de elementos da tupla.

É possível também trocar o valor de variáveis, conforme apresentado a seguir:

```
>>> x, y = 1, 3
>>> x, y = y, x
>>> x
3
>>> y
1
```

Tuplas com um elemento

Python apresenta uma sintaxe especial para criação de tuplas com apenas um elemento. O exemplo apresentado a seguir não cria uma tupla com um elemento, mesmo com o uso de () parênteses.

```
>>> t1 = (10)
>>> t1
10
```

Portanto, em Python, para criar uma tupla com um elemento deve-se, obrigatoriamente, acrescentar uma vírgula após o elemento que poderá ou não fazer uso de () parênteses, conforme segue:

```
>>> t2 = (10,)
>>> t2
(10,)
>>> t3 = 10,
>>> t3
(10,)
```

Tuplas vazias

A linguagem Python possibilita a criação de tuplas vazias. Porém, neste caso, o uso de parênteses é obrigatório.

```
>>> t = ()
>>> t
()
```

10. Conjuntos (sets)

Objetivo: Apresentar as características e operações que pode ser realizadas em conjuntos de dados em Python.

Os conjuntos (**set**), em Python são estruturas que diferem das apresentadas anteriormente, pois apresentam as seguintes características:

- Implementam operações como união, intersecção e diferença
- Não admitem elementos repetidos
- Não mantêm a ordem de seus elementos

Um set pode ser criado a partir de listas, tuplas ou outra estrutura de dados enumerável.

O exemplo a seguir apresenta a criação de um set e, na sequência, a tentativa de adicionar um elemento repetido ao conjunto.

```
>>> x = set([1, 3, 5])
>>> x
{1, 3, 5}
>>> x.add(3)
>>> x
{1, 3, 5}
```

O elemento não foi adicionado (com o uso da função **add**) pois já havia outro no **set** com o mesmo valor.

O operador **in** é usado para verificar se um elemento faz parte do set:

```
>>> 1 in x
True
>>> 2 in x
False
```

União

A operação de união, realizada com o uso do operador **|**, apresenta os elementos de **x** e de **y**, sem repetições:

```
>>> x = set([1, 3, 5])
>>> y = set([3, 7])
>>> x | y
{1, 3, 5, 7}
```

Intersecção

A operação de intersecção, realizada com o uso do operador **&**, apresenta os elementos que estão presentes em **x** e em **y**:

```
>>> x = set([1, 3, 5])
>>> y = set([3, 7])
>>> x & y
{3}
```

Diferença

A operação de diferença, realizada com o uso do operador **-**, apresenta os elementos de **x** que não estão presentes em **y**:

```
>>> x = set([1, 3, 5])
>>> y = set([3, 7])
>>> x - y
{1, 5}
```

Python: Resumo das estruturas de dados

Características	List	Tuple	Dictionary	Set
Principal uso	Sequências	Sequências constantes	Elementos indexados	Unicidade dos elementos Operações de conjunto
Tamanho	Variável	Fixo	Variável	Variável
Alterações	Sim	Não	Sim	Sim
Ordem dos elementos	Fixa	Fixa	Fixa (a partir do Python 3.7)	Indeterminada
Repetição de elementos	Sim	Sim	Valores: sim Chaves: não	Não
Pesquisa	Sequencial	Sequencial	Direta por chaves	Direta por valor

11. Funções

Objetivo: Apresentar a criação e utilização de funções com a linguagem Python.

Funções são utilizadas especialmente para realizar tarefas específicas em um trecho do programa. Cada função recebe um nome e isso possibilita sua reutilização em outras partes do programa.

A instrução **def** deve ser utilizada para criar uma nova função. O exemplo a seguir apresenta a criação de uma função denominada **soma** que recebe dois valores (**x** e **y**) como parâmetros e imprime a **soma** dos dois.

```
def soma (x, y):  
    print(x + y)
```

```
>>> soma(5, 10)  
15
```

O resultado da soma foi obtido através do comando **print**, a função apresentada não retornou valores ao ambiente que a chamou. Para que uma função retorne valores deve ser utilizada a instrução **return**.

O exemplo a seguir utiliza a instrução **return** para retornar ao ambiente o valor da **soma** de dois números.

```
def soma (x, y):  
    return x + y
```

```
>>> print(soma(5, 10))  
15
```

A função apresentada a seguir calcula a média dos valores de da lista (**L**).

```
def media(L):  
    total = 0  
    for n in L:  
        total += n  
    return total / len(L)
```

```
>>> L = (1, 2, 3)  
>>> print(media(L))  
2.0
```

Uma função também pode retornar valores booleanos (**True** ou **False**). O exemplo a seguir retornará **True** se o número for par e **False** se não for par.

```
def npar(x):  
    return x % 2 == 0
```

```
>>> print(npar(4))  
True  
>>> print(npar(5))  
False
```

Retornando mais de um valor

Uma função em Python pode retornar mais de um valor. A função **operacoes** apresentada a seguir retorna dois valores: a **soma** e o **produto** de dois números inteiros.

```
def operacoes(x, y):  
    return x + y, x * y  
soma, produto = operacoes (3, 5)
```



```
print("Soma: ", soma)
print("Produto: ", produto)
Soma: 8
Produto: 15
```

Funções recursivas

Uma função é denominada recursiva quando dentro do seu código existe uma chamada para si mesma. Um exemplo clássico desse tipo de função é o cálculo do fatorial. A função fatorial apresentada a seguir chama a si mesma quantas vezes forem necessárias para calcular o fatorial do número usado como parâmetro.

```
def fatorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * fatorial(n - 1)

>>> print(fatorial(3))
6
```

Variáveis locais e globais

Python, como outras linguagens de programação permite ao programador trabalhar com variáveis **locais** (internas a uma função) ou **globais** (externas à função). O exemplo a seguir apresenta na prática a visibilidade ou escopo de variáveis. Embora não seja um procedimento recomendável, é utilizado o mesmo nome (**x**) para duas variáveis, sendo uma **local** à função **teste()** e outra **global** para demonstrar o escopo de cada variável.

```
x = 10
def teste():
    x = 5
    print(x)

>>> print(x)
10
>>> teste()
5
>>> print(x)
10
```

Parâmetros opcionais

Parâmetros opcionais são usados quando não é necessário passar todos os parâmetros para uma função. Neste caso, utiliza-se um valor previamente determinado para o parâmetro opcional, com a possibilidade de alterá-lo, caso necessário.

```
def soma(a, b, c = 0):
    s = a + b
    if c:
        s = a + b + c
    print(s)

>>> soma(1,3)
4
>>> soma(1,3,5)
9
```

Os parâmetros opcionais devem sempre ser declarados por último, isto é, após a declaração dos parâmetros obrigatórios.

Nomeando parâmetros

A linguagem Python permite atribuir nomes ou identificadores aos parâmetros, conforme observado nos exemplos anteriores. A função a seguir apresenta dois parâmetros, denominados **a** e **b**.

```
def soma (a, b):  
    s = a + b  
    print (s)
```

Quando a função é chamada, caso não se utilize os nomes dos parâmetros, os valores devem ser passados na mesma ordem em que foram declarados na função.

```
>>> soma(1, 3)  
4
```

No entanto, caso se utilize os nomes dos parâmetros, os valores podem ser passados em qualquer ordem.

```
>>> soma(a=1, b=3)  
4  
>>> soma(1,3,5)  
9
```

Funções como parâmetros

Uma função pode ter vários parâmetros, que podem ser objetos, variáveis (do mesmo tipo de dados ou diferentes) e funções. Funções que podem aceitar outras funções como parâmetros também são chamadas de **funções de ordem superior**. No exemplo a seguir, a função **imprime** recebe uma função como parâmetro.

```
def maiusculas(texto):  
    return texto.upper()  
def minusculas(texto):  
    return texto.lower()  
def imprime(funcao):  
    # armazenando a função em uma variável  
    mensagem = funcao("Data is our business!")  
    print(mensagem)  
  
>>> imprime(maiusculas)  
DATA IS OUR BUSINESS!  
>>> imprime(minusculas)  
data is our business!
```

Funções com número indeterminado de parâmetros

O Python possibilita, através do uso de listas, criar funções com um número indeterminado de parâmetros. A função **soma**, apresentada a seguir, pode receber um número indeterminado de parâmetros.

```
def soma(*args):  
    s = 0  
    for n in args:  
        s += n  
    return s
```

```
>>> soma ()
0
>>> soma (1)
1
>>> soma (1,3)
4
>>> soma (1,3,5)
9
```

Funções Lambda

Python permite a criação de funções anônimas usando um recurso chamado função **lambda**. Podem ter qualquer número de argumentos, como uma função normal. As funções lambda geralmente não apresentam mais do que uma linha. O corpo das funções lambda é muito pequeno e consiste em apenas uma expressão. O resultado da expressão é o valor quando o lambda é aplicado a um argumento. Além disso, não há necessidade de nenhuma instrução **return** na função lambda.

```
>>> r = lambda x, y: x * y
>>> r(2,3)
6
```

Modularização

Sistemas desenvolvidos em Python podem fazer uso de várias funções que podem ser usadas mais de uma vez em diversas partes do programa. Python permite, portanto, criar módulos, isto é, armazenar funções de uso frequente em outros arquivos. Todo arquivo com o sufixo **.py** é um módulo e pode ser importado com o comando **import**.

O exemplo a seguir apresenta dois programas: **adicao.py** e **usa_adicao.py**. Usando o comando **import** é possível chamar a função **soma** definida em **adicao.py**. O nome do módulo (**adicao**) deve ser declarado antes do nome da função (**soma**) e devem ser separados por **.** (ponto): **adicao.soma**.

```
# arquivo adicao.py
def soma (*args):
    s = 0
    for n in args:
        s += n
    return s

# arquivo usa_adicao.py
import adicao
print(adicao.soma(1, 3, 5))
```

12. Exceções

Objetivo: Apresentar as exceções predefinidas e as definidas pelo usuário para tratar erros que podem ocorrer durante a execução de um programa.

Exceções são erros que ocorrem durante a execução de um programa. Quando devidamente tratadas podem evitar que um programa termine de forma anormal.

Tratamento de exceções

Quando ocorre uma exceção é possível tratá-las com os comandos contidos nas cláusulas `try`, `except`, `else` e `finally` que seguem a ordem apresentada a seguir:

```
try
    try:
        comandos_1 ...
    except ([tipo_exceção1, ...]) [as argumento]:
        comandos_2
[else:
    comandos_3]
[finally:
    comandos_4]
```

Onde:

- `comandos_1`: Comandos que podem gerar uma exceção
- `tipo_exceção_1, ...`: Exceções que se pretende capturar
- `argumento`: Argumento que terá a informação sobre a causa da exceção
- `comandos_2`: Comandos a serem executados caso ocorra uma das exceções
- `comandos_3`: Comandos a serem executados caso não ocorra uma exceção
- `comandos_4`: Comandos a serem executados caso ocorra ou não uma exceção

Exemplo:

```
lista = ["Janeiro", "Fevereiro", "Março"]
n = int(input("Digite o número do mês: 1 a 3: "))
n = n - 1
try:
    print(lista[n])
    print("Não ocorreu exceção")
except:
    print("Ocorreu uma exceção.")
print("Continuação do programa ...")
```

Caso 1: Exceção não disparada:

```
Digite o número do mês: 1 a 3: 3
Março
Não ocorreu exceção
Continuação do programa ...
```

Caso 2: Exceção disparada:

Digite o número do mês: 1 a 3: **4**

Ocorreu uma exceção.

Continuação do programa ...

Exceções predefinidas

Python tem várias exceções predefinidas: `IndexError`, `ValueError`, `TypeError`, `ZeroDivisionError` e outras, que são disparadas quando ocorre algum erro em um programa.

O exemplo a seguir apresenta a exceção predefinida **`ZeroDivisionError`**, que será disparada quando o usuário tentar realizar a divisão de um número por 0 (zero).

```
import math
dividendo = int(input("Digite o dividendo: "))
divisor = int(input("Digite o divisor: "))
quociente = 0
try:
    quociente = dividendo / divisor
except(ZeroDivisionError) as arg:
    print("Exceção: ", arg)
else:
    print("Resultado: ", quociente)
```

Digite o dividendo: 10

Digite o divisor: 0

Exceção: division by zero

Exceções definidas pelo usuário

Exceções também podem ser definidas pelo usuário (programador). Neste caso, deverá criar uma nova classe de exceção que, direta ou indiretamente, será derivada da classe **`Exception`** do Python.

O exemplo a seguir utiliza exceções criadas pelo usuário para apresentar mensagens de erro caso o valor digitado seja inferior (**`NumeroNegativoError`**) ou superior (**`NumeroMaiorError`**) ao intervalo determinado.

```
class MyException(Exception):
    pass
class NumeroNegativoError(MyException):
    pass
class NumeroMaiorError(MyException):
    pass
while True:
    try:
        n = int(input("Digite um número de 0 a 10: "))
        if n < 0:
            raise NumeroNegativoError
        else:
            if n > 10:
                raise NumeroMaiorError
            break
    except NumeroNegativoError:
        print("Número negativo")
    except NumeroMaiorError:
        print("Número maior que 10")
print(f"Você digitou: {n}")
```

13. Arquivos

Objetivo: Apresentar a criação de arquivos de texto com a linguagem Python para persistência de dados.

Arquivos permitem armazenar permanentemente os dados. Um arquivo é acessado por nome e, através de programas criados em Python ou outras linguagens de programação é possível ler e escrever dados.

Para acessar um arquivo é preciso primeiro abri-lo com a função **open**. A função **open** trabalha com dois parâmetros: **nome do arquivo** e **modo**. O parâmetro **modo** indica a forma como o arquivo será aberto ou as operações a serem realizadas, conforme segue:

Modo	Operações
r	leitura
w	escrita
a	escrita preservando o conteúdo, se existir
b	modo binário
+	atualização (leitura e escrita)

Os modos apresentados podem ser combinados: **r+**, **w+**, **a+**, etc.

A função **open** retorna um objeto do tipo **file**, que será utilizado para ler e escrever os dados. O método **write** escreve no arquivo e o método **read** realiza a leitura do seu conteúdo. O fechamento do arquivo com a função **close** é importante, pois libera os recursos do computador que já não precisam ser utilizados.

O exemplo a seguir cria um arquivo denominado **numeros.txt** e **escreve** nele os números de 0 a 9.

```
# gravar.py
arquivo = open("numeros.txt", "w")
for linha in range(1, 10):
    arquivo.write(f"{linha}\n")
arquivo.close()
```

O próximo exemplo realiza a leitura do conteúdo do arquivo **numeros.txt**. O método **readlines** gera uma lista em que cada elemento é uma linha do arquivo.

```
# ler.py
arquivo = open("numeros.txt", "r")
for linha in arquivo.readlines():
    print(linha)
arquivo.close()
```

Caso o arquivo **grava.py** seja executado novamente, os valores anteriores, serão **sobregravados** e, portanto, perdidos. No entanto é possível acrescentar linhas no mesmo arquivo **numeros.txt**, sem perder os dados anteriormente gravados. Neste caso o modo de abertura do arquivo deve ser substituído de **"r"** para **"a"**, conforme segue:

```
arquivo = open("numeros.txt", "a")
for linha in arquivo.readlines():
    print(linha)
arquivo.close()
```

14. Programação Orientada a Objetos

Objetivo: Apresentar a criação e instanciação de classes e outros conceitos de programação orientada a objetos com Python.

Python foi desenvolvido sob o paradigma da programação orientada a objetos, portanto criar e usar classes e objetos torna-se muito prático.

Terminologia POO (Programação Orientada a Objetos)

Classe: Protótipo de um objeto que define um conjunto de atributos (variáveis de classe e variáveis de instância) e métodos.

Variável de classe: Variável que é compartilhada por todas as instâncias de uma classe. Variáveis de classe são definidas dentro de uma classe, mas fora de qualquer um dos métodos da classe. Variáveis de classe não são usadas com tanta frequência quanto as variáveis de instância.

Membro de dados: Variável de classe ou variável de instância que contém dados associados a uma classe e seus objetos.

Sobrecarga de método: Atribuição de mais de um comportamento a um método específico. A operação executada varia de acordo com os tipos de objetos ou argumentos envolvidos.

Variável de instância: Variável que é definida dentro de um método e pertence apenas à instância atual de uma classe.

Herança: Transferência das características de uma classe para outras classes que são derivadas dela.

Instância: Objeto individual de uma determinada classe. Um objeto que pertence a uma classe Cliente, por exemplo, é uma instância da classe Cliente.

Instanciação: Criação de uma instância de uma classe.

Método: Tipo especial de função que é definido em uma classe.

Objeto: Instância única de uma estrutura de dados definida por sua classe. Um objeto compreende membros de dados (variáveis de classe e variáveis de instância) e métodos.

Sobrecarga do operador: Atribuição de mais de uma função a um determinado operador.

Criando uma classe

A palavra reservada **class** deve ser usada para criar uma classe. Nomes de classes devem começar com maiúsculas. Não pode haver espaços em nomes de classes. Adota-se, portanto, a notação conhecida como CamelCase, variações entre letras maiúsculas e minúsculas, para compor nomes de classes. Exemplo: MinhaClasse.

O exemplo a seguir cria a classe **MinhaClasse** com uma única propriedade (atributo) **x**.

```
class MinhaClasse:
    x = 10
# cria p1, uma instância da classe MinhaClasse
p1 = MinhaClasse()
print(p1.x)
```

O método `__init__()`

O exemplo anterior representa classe e objeto em sua forma mais simples e não são realmente úteis em aplicativos da vida real.

Na prática, todas as classes Python possuem um método chamado `__init__`, que é sempre executado quando a classe está sendo iniciada. É usado para atribuir valores às propriedades do objeto ou outras operações necessárias quando o objeto está sendo criado.

O método `__init__` é chamado automaticamente toda vez que a classe está sendo usada para criar um novo objeto.

O exemplo a seguir cria uma classe denominada **Cliente** com dois atributos: **nome** e **telefone**. A instância da classe Cliente (**c1**) atribui valores aos dois atributos.

```
# clientes.py
class Cliente:
    def __init__(self, nome, telefone):
        self.nome = nome
        self.telefone = telefone

c1 = Cliente("JOHN SMITH", "5555-1111")
print(c1.nome)
print(c1.tlefone)
```

Classes também podem conter métodos, isto é, funções que pertencem à classe.

A seguir, o método **myfunc** é adicionado à classe **Cliente** anteriormente criada.

```
# clientes.py
class Cliente:
    def __init__(self, nome, telefone):
        self.nome = nome
        self.telefone = telefone
    def myfunc(self):
        print("Nome: " + self.nome)

c1 = Cliente("JOHN SMITH", "5555-1111")
c1.myfunc()
```

A seguir, é criada uma nova classe denominada **Conta** que recebe **clientes** (lista de objetos), **nrconta** (string) e **saldo** (parâmetro opcional com 0 como padrão). A classe também apresenta três métodos: **extrato**, **deposito** e **saque**.

```
# contas.py
class Conta:
    def __init__(self, clientes, nrconta, saldo = 0):
        self.clientes = clientes
        self.nrconta = nrconta
        self.saldo = saldo
    def extrato(self):
        print(f"Conta Nr: {self.nrconta} Saldo: {self.saldo:10.2f}")
    def deposito(self, valor):
        self.saldo += valor
    def saque(self, valor):
        if self.saldo >= valor:
            self.saldo -= valor
```


Herança

Herança é um conceito do paradigma da orientação à objetos que determina que uma classe "filha" herde atributos e métodos de uma classe "pai" ou "super classe" e, desta forma, evita repetições de código. A classe **ContaEspecial**, a seguir, herda da classe **Conta**, todos os seus atributos e adiciona um outro atributo opcional: **limite** (inicializado em 0). O método **saque** foi alterado para somar ao **saldo** o **limite** estabelecido para conta.

```
# ... continua no arquivo contas.py
class ContaEspecial(Conta):
    def __init__(self, clientes, nrconta, saldo = 0, limite = 0):
        Conta.__init__(self, clientes, nrconta, saldo)
        self.limite = limite
    def saque(self, valor):
        if self.saldo + self.limite >= valor:
            self.saldo -= valor
```

A seguir, é criado um novo arquivo, **teste.py**, para testar as classes **Cliente**, **Conta** e **ContaEspecial**.

```
# teste.py
from clientes import Cliente
from contas import Conta, ContaEspecial
cliente1 = Cliente("JOHN SMITH", "5555-1111")
cliente2 = Cliente("MARY CLARK", "5555-2222")
conta1 = Conta([cliente1], 1, 2000)
conta2 = ContaEspecial([cliente2], 2, 1000, 5000)
conta1.deposito(500)
conta1.saque(200)
conta2.deposito(300)
conta2.saque(100)
conta1.extrato()
conta2.extrato()
print(cliente1.nome)
print(cliente2.nome)
```

15. Banco de dados: SQLite

Objetivo: Apresentar como realizar a conexão com bancos de dados para criação de tabelas, consultas, inserção, alteração e eliminação de dados utilizando Python e SQL.

O banco de dados SQLite é um gerenciador de banco de dados leve, completo e já vem pré instalado com o interpretador Python. O SQLite não precisa de um servidor dedicado e pode ser utilizado pelo Python por meio do módulo predefinido **sqlite3**.

Criação de bancos de dados e tabelas

O exemplo a seguir utiliza a SQL, linguagem padrão dos bancos de dados relacionais, para criar o database (banco de dados) **agenda.db**. Depois cria uma tabela denominada **contatos** com duas colunas **nome** e **telefone**. Por último, insere uma linha na tabela com um nome ("John Smith") e um telefone ("5555-1111").

```
import sqlite3
conexao = sqlite3.connect("agenda.db")
cursor = conexao.cursor()
cursor.execute('''
    create table contatos (
        nome text,
        telefone text )
''')
cursor.close()
conexao.close()
```

Comentários:

```
import sqlite3
```

Importa o módulo predefinido **sqlite3**.

```
conexao = sqlite3.connect("agenda.db")
```

Abre uma conexão com o banco de dados **agenda.db**.

```
cursor = conexao.cursor()
```

Cria um objeto **cursor** por meio do método **cursor()** para enviar informações ao banco de dados.

```
cursor.execute('' create table ...
```

Utiliza o método **execute** do objeto **cursor** para criar a tabela **contatos**.

```
cursor.close()
```

Fecha o cursor.

```
conexao.close()
```

Fecha a conexão com o banco de dados.

Tipos de dados

O SQLite não dispõe de muitos tipos de dados. A tabela a seguir apresenta a relação entre os tipos de dados do Python e do SQLite.

Python	SQLite
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

Inserção de linhas em tabelas

O comando SQL **insert** insere novas linhas em tabelas previamente criadas.

O exemplo a seguir apresenta a inserção de uma linha na tabela **contatos** anteriormente criada.

```
import sqlite3
conexao = sqlite3.connect("agenda.db")
cursor = conexao.cursor()
cursor.execute('''
    insert into contatos (nome, telefone)
    values(?, ?)''', ("JOHN SMITH", "5555-1111"))
conexao.commit()
cursor.close()
conexao.close()
```

Comentários:

```
cursor.execute(''' insert into ...
```

Utiliza o método **execute** do objeto **cursor** para inserir uma linha na tabela **contatos**.

```
conexao.commit()
```

Confirma a transação, isto é, a inserção da linha no banco de dados.

É possível também inserir várias linhas de uma só vez através do método **executemany** do objeto **cursor**, que permite executar um comando SQL aplicado, por exemplo, a uma lista, conforme exemplo a seguir.

```
import sqlite3
lista = [("MARY CLARK", "5555-2222"), ("NOAH BAKER", "5555-3333")]
conexao = sqlite3.connect("agenda.db")
cursor = conexao.cursor()
cursor.executemany('''
    insert into contatos (nome, telefone)
    values(?, ?)''', lista)
conexao.commit()
cursor.close()
conexao.close()
```

Consultas

O comando SQL **select** deve ser utilizado para realizar consultas nas tabelas de um banco de dados.

O exemplo a seguir apresenta a **inserção** de uma linha na tabela **contatos** anteriormente criada.

```
import sqlite3
conexao = sqlite3.connect("agenda.db")
cursor = conexao.cursor()
cursor.execute("select * from contatos")
linhas = cursor.fetchall()
for linha in linhas:
    print(linha)
cursor.close()
conexao.close()
```

Comentários:

```
linhas = cursor.fetchall()
```

Chama o método **fetchall** do objeto **cursor** para buscar os dados.

```
for linha in linhas:  
    print(linha)
```

Faz um loop com o cursor e processa cada linha individualmente.

Eliminação de linhas de tabelas

O comando SQL **delete** deve ser utilizado para eliminar linhas de tabelas de um banco de dados.

O exemplo a seguir apresenta a **exclusão** de uma linha na tabela **contatos** anteriormente criada.

```
import sqlite3  
conexao = sqlite3.connect("agenda.db")  
cursor = conexao.cursor()  
cursor.execute('delete from contatos where nome = ?', ("JOHN SMITH",))  
conexao.commit()  
print("Linhas eliminadas: ", cursor.rowcount)  
cursor.close()  
conexao.close()
```

Comentários:

```
cursor.execute('delete from
```

Elimina as linhas conforme o critério especificado na cláusula **where**.

```
print("Linhas eliminadas: ", cursor.rowcount)
```

O atributo **rowcount**, do objeto cursor, retorna o número de linhas eliminadas.

Atualização de dados nas tabelas

O comando SQL **update** deve ser utilizado para atualizar dados de linhas das tabelas.

O exemplo a seguir apresenta a **atualização** de uma linha na tabela **contatos** anteriormente criada.

```
import sqlite3  
conexao = sqlite3.connect("agenda.db")  
cursor = conexao.cursor()  
cursor.execute('update contatos set telefone = "5555-8888"  
    where nome = ?', ("MARY CLARK",))  
conexao.commit()  
cursor.close()  
conexao.close()
```

Comentários:

```
cursor.execute('update ...
```

Atualiza as linhas conforme o critério especificado na cláusula **where**.

16. Banco de dados: MongoDB

Objetivo: Apresentar a criação de databases e coleções no banco de dados MongoDB e a manipulação de dados em documentos usando a linguagem Python.

MongoDB

O MongoDB armazena dados em documentos do tipo BSON (versão binária do JSON), o que torna o banco de dados muito flexível e escalonável. O download do MongoDB está disponível em:

<https://www.mongodb.com>.

PyMongo

O Python precisa do driver **PyMongo** para acessar o banco de dados MongoDB. O **PyMongo** é instalado através do gerenciador de pacotes, conforme segue:

Program Files\Python\Scripts>python -m pip install pymongo

Para testar se a instalação foi bem-sucedida ou para verificar se o **pymongo** já está instalado, pode-se criar um arquivo com conteúdo apresentado a seguir. Se o código for executado sem erros, o **pymongo** está instalado e pronto para ser usado.

```
# teste_mongodb.py
import pymongo
```

Database, collection e document

O primeiro passo para criar um banco de dados no MongoDB, é criar um objeto **MongoClient** e especificar uma URL de conexão com o endereço IP e o nome do banco de dados.

O exemplo a seguir cria um banco de dados denominado **mydatabase**. O MongoDB cria o banco de dados, se ele não existir, e faz a conexão com ele. A seguir é criada a coleção **clientes** e o primeiro documento é inserido nessa coleção.

```
# mydatabase1.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
mydict = { "_id": 1, "nome": "JOHN SMITH", "telefone": "5555-1111" }
x = mycol.insert_one(mydict)
```

O MongoDB não cria um banco de dados até que nele seja criada uma coleção e um documento.

O comando apresentado a seguir verifica se a coleção foi criada.

```
>>> print(mydb.list_collection_names())
['customers']
```

Inserindo vários documentos em uma coleção

O exemplo a seguir usa o método **insert_many** para inserir vários documentos em uma coleção.

```
# mydatabase2.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
```

```
mylist = [
    { "_id": 2, "nome": "MARY CLARK", "telefone": "5555-2222"},
    { "_id": 3, "nome": "NOAH BAKER", "telefone": "5555-3333"},
    { "_id": 4, "nome": "KATE BROWN", "telefone": "5555-4444"}
]
x = mycol.insert_many(mylist)
# lista os valores _id dos documentos inseridos:
print(x.inserted_ids)
```

Consultas

Assim como a instrução SELECT é usada para localizar dados em uma tabela em um banco de dados relacional, no MongoDB, os métodos **find** e **find_one** são usados para localizar dados em uma coleção.

O método **find_one** retorna a primeira ocorrência na seleção.

```
# mydatabase3.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
x = mycol.find_one()
print(x)
```

O método **find** retorna todas as ocorrências na seleção. O primeiro parâmetro do método **find** é um objeto de consulta.

No exemplo apresentado a seguir é usado um objeto de consulta vazio, que seleciona todos os documentos da coleção. Se nenhum parâmetro for apresentado no método **find** o resultado será o mesmo que SELECT * dos bancos relacionais.

```
# mydatabase4.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
for x in mycol.find():
    print(x)
```

O segundo parâmetro do método **find** é um objeto que descreve quais campos incluir no resultado. Este parâmetro é opcional e, se omitido, todos os campos serão incluídos no resultado.

O exemplo apresentado a seguir retorna apenas os nomes e telefones, não os **_ids**:

```
# mydatabase5.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
for x in mycol.find({}, { "_id": 0, "nome": 1, "telefone": 1 }):
    print(x)
```

Não se pode especificar os valores 0 e 1 no mesmo objeto (exceto se um dos campos for o campo **_id**). Caso seja especificado um campo com o valor 0, todos os outros campos receberão o valor 1 e vice-versa.

É possível também filtrar o resultado usando um objeto de consulta. O primeiro argumento do método **find** é um objeto de consulta usado para limitar a pesquisa.

O exemplo apresentado a seguir usa um filtro para encontrar o(s) documento(s) com o telefone "5555-1111".

```
# mydatabase5.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
myquery = { "telefone": "5555-1111" }
mydoc = mycol.find(myquery)
for x in mydoc:
    print(x)
```

Para fazer consultas avançadas, pode-se usar modificadores como valores no objeto de consulta. Por exemplo, para localizar os documentos em que o campo "nome" começa com a letra "J" ou superior (alfabeticamente), usa-se o modificador maior que: {"\$gt": "J"}, conforme segue.

```
# mydatabase6.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
myquery = { "nome": { "$gt": "J" } }
mydoc = mycol.find(myquery)
for x in mydoc:
    print(x)
```

Expressões regulares também podem ser usadas como um modificador. Porém, só podem ser usadas para consultar strings. Para encontrar apenas os documentos onde o campo "nome" começam com a letra "J", deve-se usar a expressão regular {"\$regex": "^J"}, conforme segue.

```
# mydatabase7.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
myquery = { "nome": { "$regex": "^J" } }
mydoc = mycol.find(myquery)
for x in mydoc:
    print(x)
```

O método **limit** limita a quantidade de registros que serão retornados em um consulta. O método **limit** recebe um parâmetro, um número inteiro, que define quantos documentos deve retornar.

O exemplo a seguir limita o resultado para retornar apenas 5 (cinco) documentos.

```
# mydatabase8.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
myresult = mycol.find().limit(5)
```

```
# imprime o resultado
for x in myresult:
    print(x)
```

Ordenação

O método **sort** classifica o resultado de uma consulta em ordem crescente ou decrescente. O método recebe um parâmetro para **fieldname** e um parâmetro para **direção** (ascendente é a direção padrão).

O exemplo a seguir classifica o resultado em ordem alfabética por **nome** em ordem **crescente**.

```
# mydatabase9.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
mydoc = mycol.find().sort("nome")
for x in mydoc:
    print(x)
```

O próximo exemplo classifica o resultado em ordem alfabética por **nome** em ordem **decrescente**.

```
# mydatabase10.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
# 1 ordem crescente (pode ser omitido, pois é padrão)
# -1 ordem decrescente
mydoc = mycol.find().sort("nome", -1)
for x in mydoc:
    print(x)
```

Atualização de documentos

O método **update_one** atualiza um documento do MongoDB. O primeiro parâmetro do método **update_one** é um objeto de consulta que define qual documento atualizar. O segundo parâmetro é um objeto que define os novos valores do documento. Se a consulta encontrar mais de um documento, apenas a primeira ocorrência será atualizada.

O exemplo a seguir atualiza o telefone de "5555-1111" para "5555-8888".

```
# mydatabase11.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
myquery = { "telefone": "5555-1111" }
newvalues = { "$set": { "telefone": "5555-8888" } }
mycol.update_one(myquery, newvalues)
# imprime "clientes" após a atualização:
for x in mycol.find():
    print(x)
```


O método **update_many** atualiza todos os documentos que atendem aos critérios da consulta.

O exemplo a seguir atualiza todos os documentos em que os nomes comecem com a letra "J".

```
# mydatabase12.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
myquery = { "nome": { "$regex": "^J" } }
newvalues = { "$set": { "telefone": "5555-0000" } }
x = mycol.update_many(myquery, newvalues)
print(x.modified_count, "documentos atualizados.")
```

Excluir documentos

O método **delete_one** exclui um documento de uma coleção. O primeiro parâmetro do método **delete_one** é um objeto de consulta que define qual documento excluir. Se a consulta encontrar mais de um documento, apenas a primeira ocorrência será excluída.

O exemplo a seguir exclui o documento com o telefone "5555-2222".

```
# mydatabase13.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
myquery = { "telefone": "5555-2222" }
mycol.delete_one(myquery)
```

O método **delete_many** exclui mais um documento de uma coleção. O primeiro parâmetro do método **delete_many** é um objeto de consulta que define quais documentos excluir.

O exemplo a seguir exclui todos os documentos onde o nome começa com a letra "J":

```
# mydatabase14.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["clientes"]
myquery = { "nome": { "$regex": "^J" } }
x = mycol.delete_many(myquery)
print(x.deleted_count, " documentos excluídos.")
```

Para excluir todos os documentos de uma coleção, deve-se passar um objeto de consulta vazio para o método **delete_many**.

O exemplo a seguir exclui todos os documentos da coleção **clientes**.

```
# mydatabase15.py
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
```



```
mycol = mydb["clientes"]  
x = mycol.delete_many({})  
print(x.deleted_count, " documentos excluídos.")
```

Excluir coleções

Usa-se o método **drop** excluir uma coleção no Mongo DB. O método **drop** retorna **true** se a coleção for excluída com sucesso e **false** se a coleção não existir.

O exemplo a seguir exclui a coleção **clientes**.

```
# mydatabase16.py  
import pymongo  
myclient = pymongo.MongoClient("mongodb://localhost:27017/")  
mydb = myclient["mydatabase"]  
mycol = mydb["clientes"]  
mycol.drop()
```