

Python

Professora Miyuki - 2024



História

- A linguagem Python foi desenvolvida em 1991 pelo holandês Guido van Rossum.
- [Welcome to Python.org](https://www.python.org)



História



História

- Atualmente a propriedade intelectual da linguagem é mantida pela *Python Software Foundation*.
[https://www.python.org/psf/.](https://www.python.org/psf/)
Trata-se de uma corporação sem fins lucrativos a fim de promover, proteger e evoluir a linguagem.



Definição

- Linguagem de Alto Nível: Nível de abstração elevado (mais próxima à linguagem humana e mais distante da linguagem da máquina), prioriza a legibilidade de códigos utilizando **indentação**.
- Interpretada: Executado por um **interpretador** e depois pelo sistema operacional/processador.
- Imperativa: Execução em ações.
- Orientada a objetos: Abstração digital do mundo real, interação entre unidades denominadas de objetos.
- Suporta scripts.
- Suporta funções.
- **Tipagem dinâmica e forte:** Verificação do tipo de dado durante a execução, reconhece diferença entre números inteiros (ints) e reais (floats).



Características

- Sintaxe elegante e simples
- Tipagem dinâmica
- Linguagem interpretada
- Identação obrigatória
- Linguagem de Alto nível
- *Case Sensitive*

Tipagem forte e dinâmica

- Python é uma linguagem de tipagem forte, ou seja, não faz conversões automáticas entre tipos não compatíveis. A Variável assume o tipo de acordo com o valor atribuído.

```
>>> a = -5  
>>> b = 12.34  
>>> a + b  
7.34  
>>> b / a  
-2.468
```

Interpretador

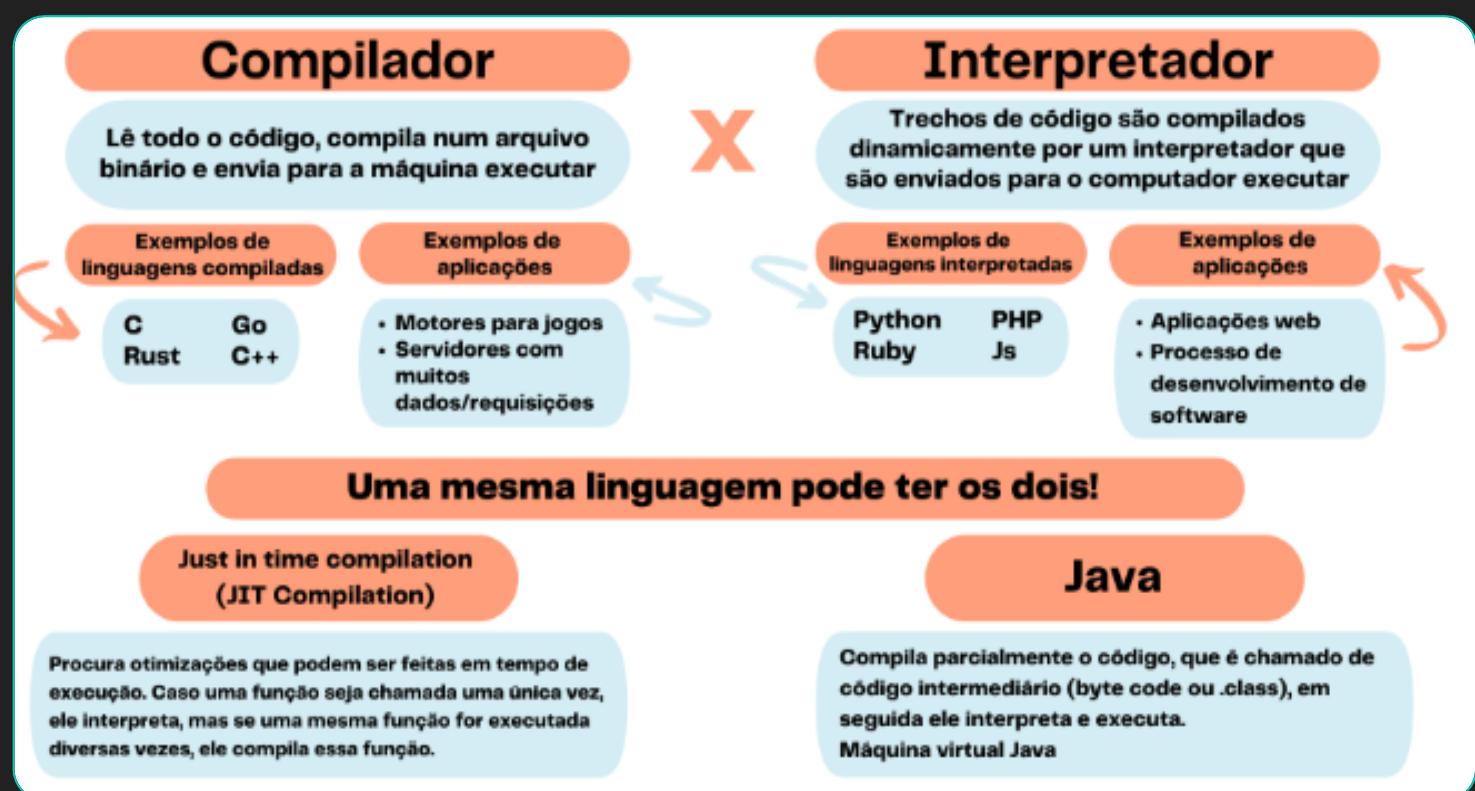
O intérprete é um programa que comprehende as instruções que você vai escrever na linguagem Python.

Sem o intérprete, o computador não vai entender as instruções e os programas não funcionam.



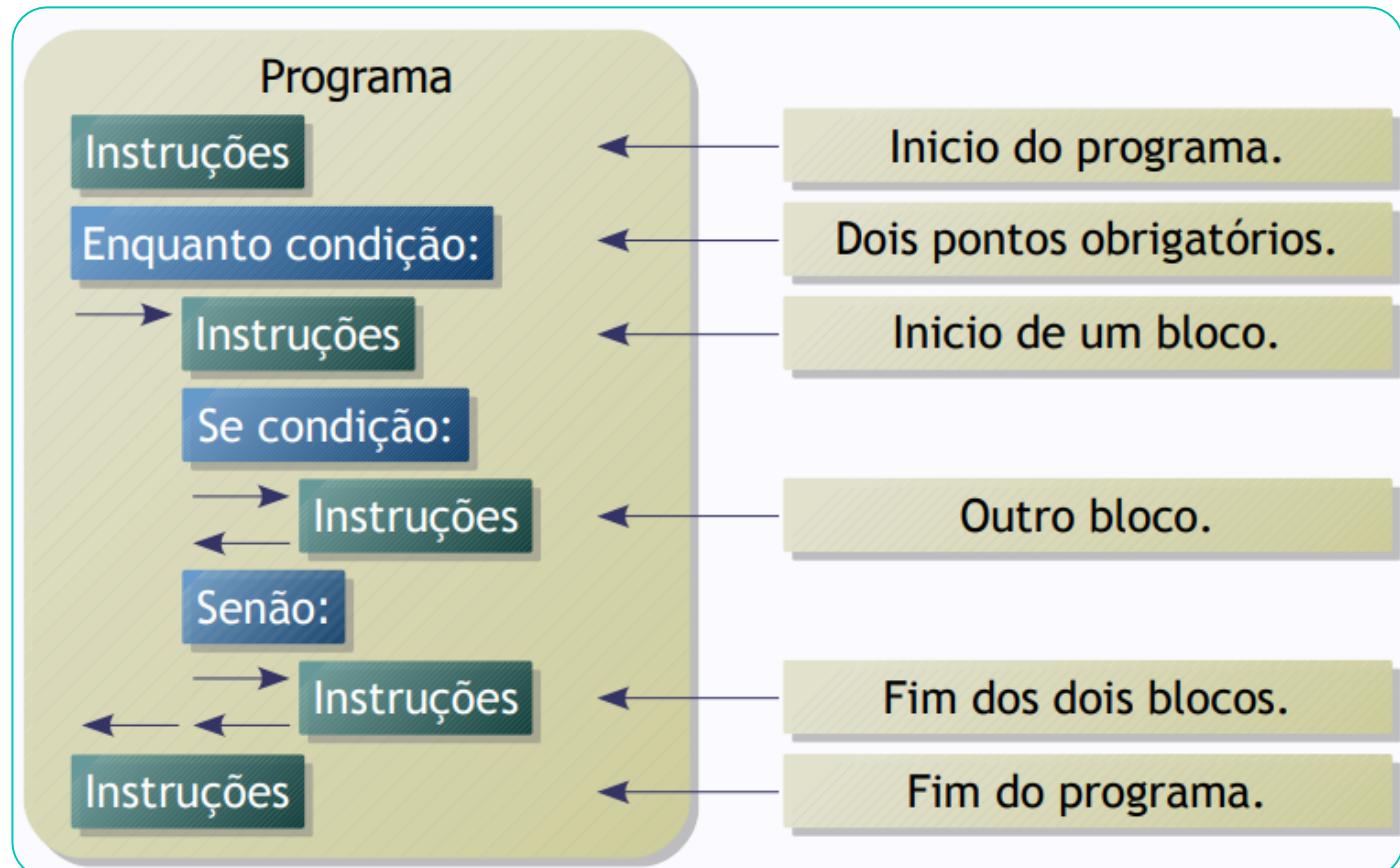
Linguagem interpretada X compilada

- Linguagens Compiladas:
C, C++, C#
- Linguagens Interpretadas:
Python, PHP, Ruby
(velocidade de execução menor)



Indentação

- Em ciência da computação, indentação é um termo aplicado ao código fonte de um programa para ressaltar ou definir a estrutura do algoritmo, aumentando assim a legibilidade do código.



Indentação

- Em Python, os blocos de código são delimitados pelo uso de indentação. A indentação deve ser constante no bloco de código, porém é considerada uma boa prática manter a consistência no projeto todo e evitar a mistura tabulações e espaços.
- # Se o resto dividindo por 3 for igual a zero: if i % 3 == 0:

Indentação

```
if (a > b && a > c)
{
    maior = a;
}
else if (b > c)
{
    maior = b;
}
else {
    maior = c;
}
printf("%d\n", maior);
```

Sem indentação

```
if (a > b && a > c)
{
    maior = a;
}
else if (b > c)
{
    maior = b;
}
else {
    maior = c;
}
printf("%d\n", maior);
```

Com indentação

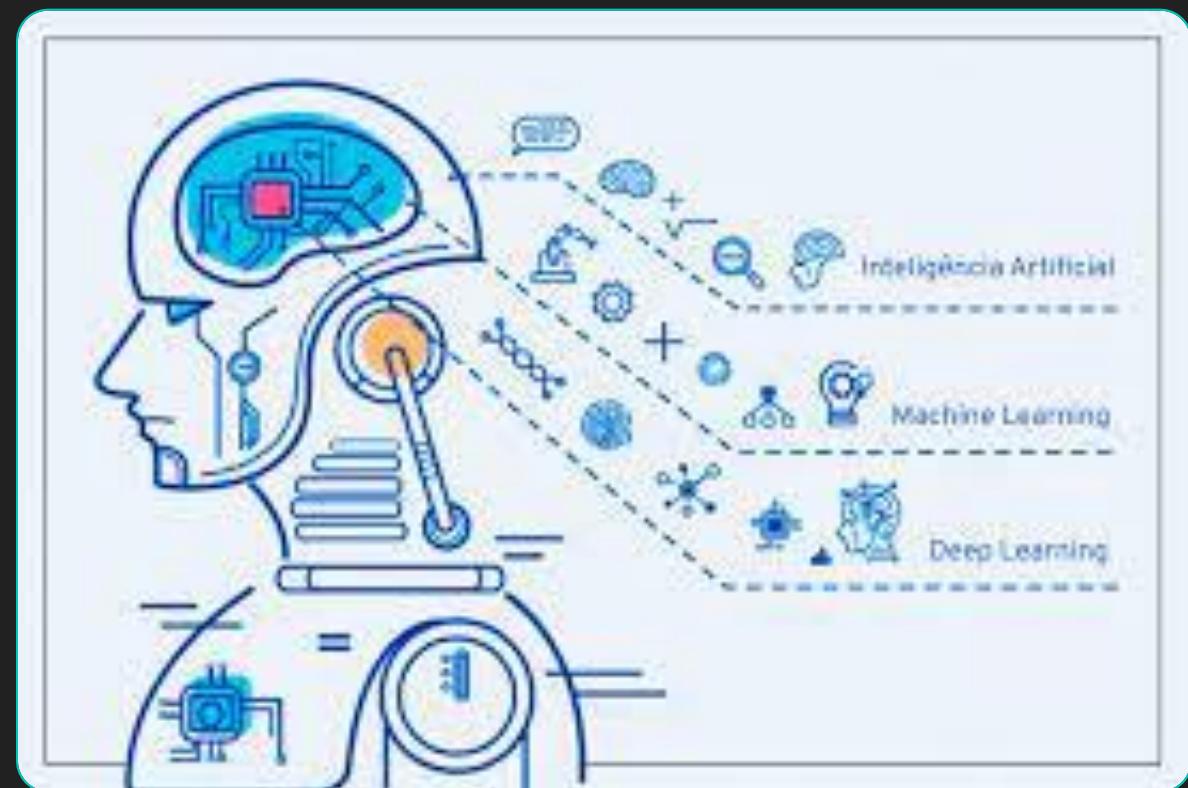
Case Sensitive

- ➊ Letras maiúsculas e minúsculas fazem diferença !
- ➋ Ex.: **Total** difere de **total** difere de **TOTAL**

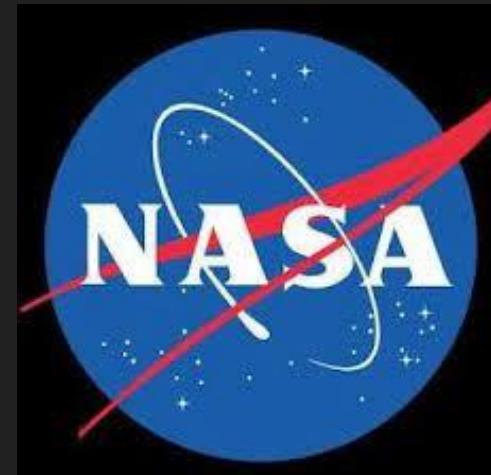


Utilização

- Machine Learning
- Inteligência Artificial
- Desenvolvimento de Jogos
- Pesquisa acadêmica e ensino



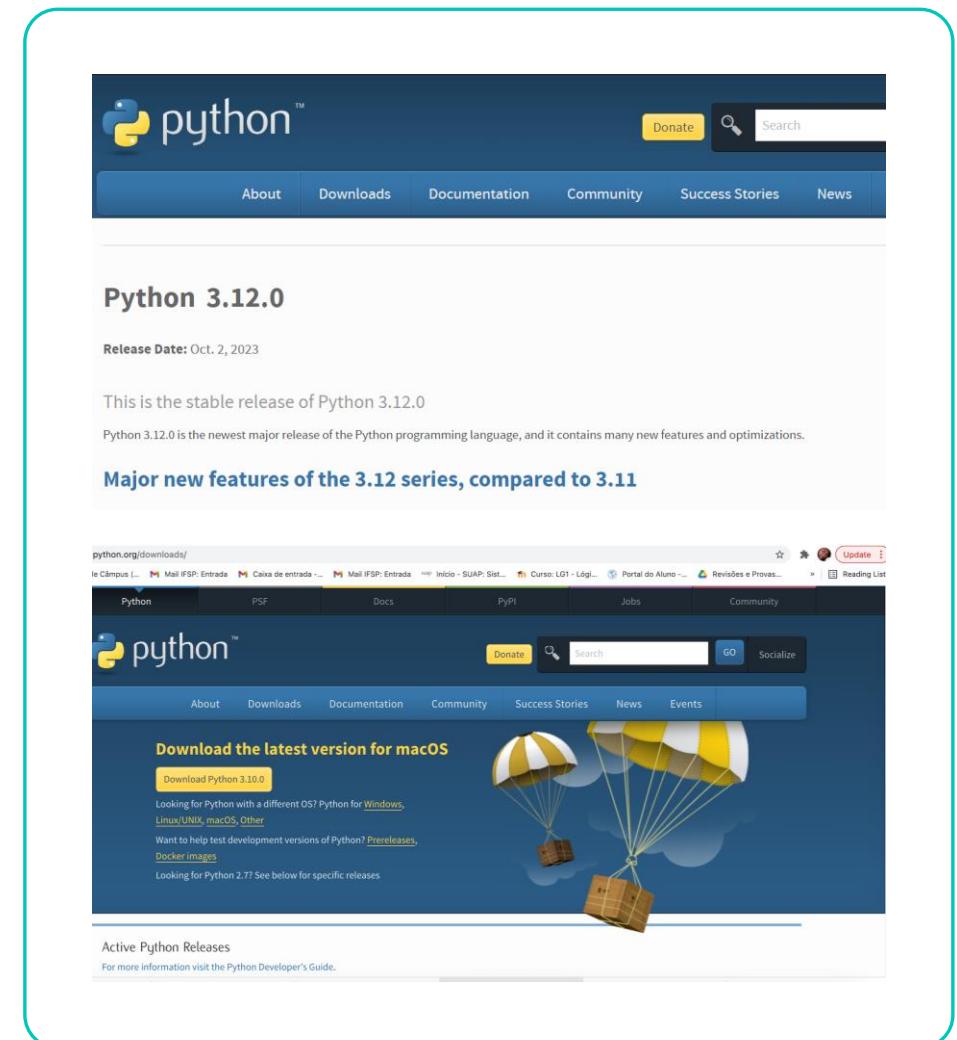
Projetos e Empresas



Testes de hardware, Serviços de Internet e vendas

Instalação

- Os sistemas operacionais Linux e MacOS já possuem o Python em seus terminais, talvez seja necessário apenas atualizar. No Windows é necessário instalar e adicionar ao path (Isso é importante para que você possa acessar o Python a partir da linha de comando.).



```
>>> print('Hello world!')  
Hello world!
```

```
>>> 8973/17  
527.8235294117648  
>>> 8973//17  
527  
>>> 2**5  
32
```

Programação

Programação

```
# Python 3: Fibonacci series up to n  
>>> def fib(n):  
>>>     a, b = 0, 1  
>>>     while a < n:  
>>>         print(a, end=' ')  
>>>         a, b = b, a+b  
>>>     print()  
>>> fib(1000)  
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610  
987
```



IDE - Integrated Development Environment

- Programas com funcionalidades e ferramentas integradas, que auxiliam os programadores no desenvolvimento de softwares.
- VS Code
- <https://colab.research.google.com/>

Palavras reservadas

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
pass	raise	return	try	while	with
yield	True	False	None		

Estruturas de Dados

Python possui classes para:

- Inteiro - int;
- Ponto flutuante - float;
- Lógico - bool;
- Sequência de Caracteres – str;
- Complexo – complex.

Tipagem dinâmica

Estruturas de Dados

- Ao contrário da maioria das outras linguagens, em Python, não é necessário declarar as variáveis que serão usadas, tampouco definir seu tipo.
- A própria sintaxe do dado a ser armazenado identifica o tipo da variável para armazená-lo.
- Por exemplo, caso necessite atribuir o valor inteiro 3 à variável A, basta digitar A=3.

Estrutura de Dados

Exemplo:

X= 4.

o ponto
indica que se
trata de float.

Estrutura de Dados

- O comando `type(var)` retorna a classe de var.

```
>>> Frase = 'Esta é uma frase'  
>>> type(Frase)  
<class 'str'>  
>>> a = 4  
>>> type(a)  
<class 'int'>  
>>> x = 2.5  
>>> type(x)  
<class 'float'>  
>>> Q = True  
>>> type(Q)  
<class 'bool'>  
>>> s = 1 + 2j  
>>> type(s)  
<class 'complex'>
```

Strings

- String é um tipo de objeto formado por uma sequência imutável de caracteres que nos permite trabalhar com textos.
- Exemplo:

```
>>> a = "Bom Dia"
```

```
>>> print a
```

```
Bom Dia
```

Strings

- Percebemos que elas são delimitadas por aspas, podemos utilizar tanto aspas duplas como as simples.
- Se utilizarmos aspas duplas, como o mostrado no exemplo anterior, podemos usar as simples para aplicações dentro do texto que estamos escrevendo, o contrário também é verdadeiro.
- Exemplo:

```
>>> b = 'O lema do governo JK era:\n “Cinquenta anos em cinco.”'
```

```
>>> print b
```

O lema do governo JK era:

“Cinquenta anos em cinco.”

OBS: \n = new line (pula linha) \t = tab (tabulação)

Tabela de manipulação de strings

len()	Retorna o tamanho da string.	len(teste) 18
capitalize()	Retorna a string com a primeira letra maiúscula	a = "python" a.capitalize() 'Python'
count()	Informa quantas vezes um caractere (ou uma sequência de caracteres) aparece na string.	b = "Linguagem Python" b.count("n") 2
startswith()	Verifica se uma string inicia com uma determinada sequência.	c = "Python" c.startswith("Py") True
endswith()	Verifica se uma string termina com uma determinada sequência.	d = "Python" d.endswith("Py") False
isalnum()	Verifica se a string possui algum conteúdo alfanumérico (letra ou número).	e = "!@#\$%" e.isalnum() False
isalpha()	Verifica se a string possui apenas conteúdo alfabetico.	f = "Python" f.isalpha() True
islower()	Verifica se todas as letras de uma string são minúsculas.	g = "pytHon" g.islower() False
isupper()	Verifica se todas as letras de uma string são maiúsculas.	h = "# PYTHON 12" h.isupper() True
lower()	Retorna uma cópia da string trocando todas as letras para minúsculo.	i = "#PYTHON 3" i.lower() "#python 3"
upper()	Retorna uma cópia da string trocando todas as letras para maiúsculo.	j = "Python" j.upper() 'PYTHON'
swapcase()	Inverte o conteúdo da string (Minúsculo / Maiúsculo).	k = "Python" k.swapcase() 'pYTHON'
title()	Converte para maiúsculo todas as primeiras letras de cada palavra da string.	l = "apostila de python" l.title() 'Apostila De Python'
split()	Transforma a string em uma lista, utilizando os espaços como referência.	m = "cana de açúcar" m.split() ['cana', 'de', 'açúcar']

Tabela de manipulação de strings

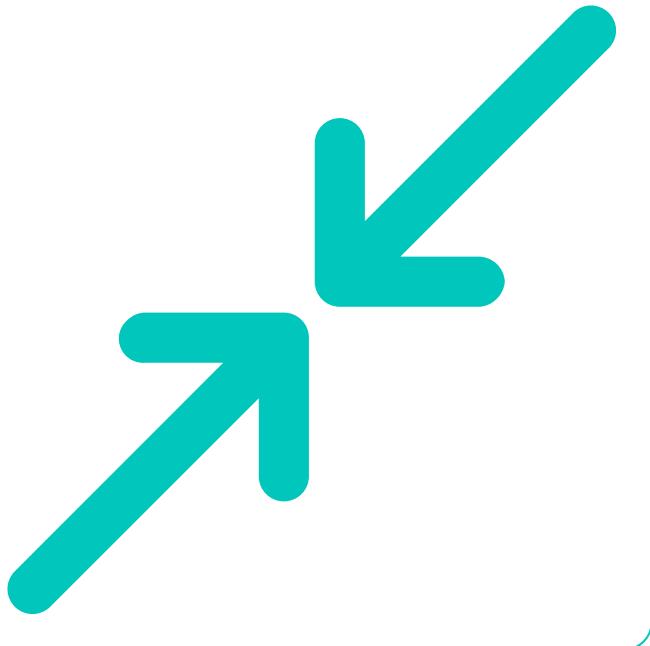
replace(S1, S2)	Substitui na string o trecho S1 pelo trecho S2.	n = "Apostila teste" n.replace("teste", "Python") 'Apostila Python'
find()	Retorna o índice da primeira ocorrência de um determinado caractere na string. Se o caractere não estiver na string retorna -1.	o = "Python" o.find("h") 3
ljust()	Ajusta a string para um tamanho mínimo, acrescentando espaços à direita se necessário.	p = " Python" p.ljust(15) ' Python '
rjust()	Ajusta a string para um tamanho mínimo, acrescentando espaços à esquerda se necessário.	q = "Python" q.rjust(15) ' Python '
center()	Ajusta a string para um tamanho mínimo, acrescentando espaços à esquerda e à direita, se necessário.	r = "Python" r.center(10) ' Python '
lstrip()	Remove todos os espaços em branco do lado esquerdo da string.	s = " Python " s.lstrip() 'Python '
rstrip()	Remove todos os espaços em branco do lado direito da string.	t = " Python " t.rstrip() ' Python '
strip()	Remove todos os espaços em branco da string.	u = " Python " u.strip() 'Python'

Comandos e Funções

- Os comandos são palavras chaves que executam certas ações e as funções são procedimentos que devolvem dados dependendo de seus argumentos.

Comandos e Funções

- As funções são similares a funções matemáticas, isto é, dependendo do argumento, o resultado é diferente.
- Quando uma função é chamada, geralmente recebe argumentos, que são passados como parâmetros, dentro de parênteses.
- As respostas da função, quando há, são chamados de retornos.



- Desta forma, temos que :
- retorno1,retorno2 = funcao(argumento1,argumento2)
- Por exemplo, ao usar a função de raiz quadrada sqrt com argumento 25, é retornado o valor 5, ou seja, para:
- >>> x = sqrt (25) A variável x receberá o valor 5.

Comandos e Funções

Importação

- A linguagem Python é conhecida pelas diversas áreas de aplicações.
- Por vezes, tais áreas necessitam de comandos ou funcionalidades específicas que não estão no Python, elas vêm de bibliotecas e precisam ser importadas, caso o usuário deseje utilizá-las.
- Para isso é utilizado o comando import.

Importação

- Por exemplo, a função sqrt para cálculo de raízes quadradas está na biblioteca math. Portanto, deverá ser impresso um erro ao utilizá-la sem importar a biblioteca.

```
>>> sqrt(9)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
```

```
>>> import math
>>> sqrt(9)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>> math.sqrt(9)
3.0
```

Bibliotecas

As bibliotecas armazenam funções pré-definidas, que podem ser utilizados em qualquer momento do programa. Em Python, muitas bibliotecas são instaladas por padrão junto com o programa. Para usar uma biblioteca, deve-se utilizar o comando import:

Exemplo: importar a biblioteca de funções matemáticas:

```
import math  
print(math.factorial(6))
```

Pode-se importar uma função específica da biblioteca:

```
from math import factorial  
print(factorial(6))
```

Bibliotecas	Função
<code>math</code>	Funções matemáticas
<code>tkinter</code>	Interface Gráfica padrão
<code>smtplib</code>	e-mail
<code>time</code>	Funções de tempo

Bibliotecas	Função
<code>urllib</code>	Leitor de RSS para uso na internet
<code>numpy</code>	Funções matemáticas mais avançadas
<code>PIL/Pillow</code>	Manipulação de imagens

Exemplos de bibliotecas (padrão e de alto nível)

```
>>> idade = input('Digite sua idade: ')
Digite sua idade:
27
>>> altura = input('Qual a sua altura: ')
Qual a sua altura:
1.75
>>> type(idade)
'int'
>>> type(altura)
'float'
```

Sabe-se que o dado digitado pelo usuário não será uma string.
Type apresenta o tipo de dado a ser manipulado.

Entrada e Saída de Dados

```
>>> x = int( input('digite um valor\n') )  
digite um valor  
2  
>>> x  
2
```

Entrada e Saída de Dados

Comando raw_input

- Através do comando raw_input podemos receber do usuário uma string.
- O tamanho da string pode ser conhecido pelo comando len().

```
>>> nome = raw_input('Digite seu nome: ')
```

```
Digite seu nome: Pedro
```

```
>>> nome
```

```
'Pedro'
```

```
>>> len(nome)
```

```
5
```

```
>>> sobrenome = raw_input('Agora digite o  
sobrenome: ')
```

```
Agora digite o sobrenome: Albuquerque
```

```
>>> print(1, 'mais', 1, 'igual a', 2)
1 mais 1 igual a 2
>>>
```

Entrada e Saída de Dados

Operador % do print

Operador %, serve para formatar as strings, basicamente são três os tipos de formatação que temos:

- %s - Serve para substituir string;
- %d - Serve para substituir números inteiros em uma frase destinada a um print;
- %f - Serve substituir floats (números em aritmética de ponto flutuante).

Observação: As três formações acima relacionadas são normalmente para aplicações em uma frase destinada a um print.

Exemplo:

```
>>> num=245.47876749  
>>> print '%.2f' %(num)  
245.48
```

Operações matemáticas com strings?

- Operações matemáticas não podem ser feitas com strings, apenas com floats e inteiros, porém se somarmos strings, Python as juntará, num processo chamado **concatenação** e se multiplicarmos uma string ela será repetida.

```
>>> nome + sobrenome 'PedroAlbuquerque'  
>>> nome*3  
'PedroPedroPedro'
```

```
>>> print('A nota de %s foi %s' % (nome, nota))  
A nota de Joao foi 10
```

Entrada e Saída de Dados - Tupla

Tipo	Operador	Descrição
Aritmético	+	Adição
Aritmético	-	Subtração
Aritmético	*	Multiplicação
Aritmético	/	Divisão
Aritmético	//	Divisão inteira
Aritmético	%	Resto da divisão
Aritmético	**	Potenciação

Operadores

Exemplo de Radiciação em Python

- Para obter a raiz n -ésima de um número basta elevá-lo por $1/n$ onde n é o índice da raiz. As regras para a obtenção de números inteiros ou pontos flutuante também se aplicam a este caso.

- Radiciação:

```
>>>8**(1.0/3.0)
```

2.0

8

```
>>> a = 5; b = 3
>>> a + b
8
>>> a/b
1.6666666666666667
>>> a//b
1
```

Operadores

Tipo	Operador	Descrição
Relacional	<code>==</code>	Igual a
Relacional	<code>!=</code>	Diferente de
Relacional	<code><></code>	Diferente de
Relacional	<code>></code>	Maior que
Relacional	<code>>=</code>	Maior ou igual a
Relacional	<code><</code>	Menor que
Relacional	<code><=</code>	Menor ou igual a

Operadores

Operadores Lógicos

Os operadores lógicos são: and, or, not, is e in.

Relacionais:

- and: retorna verdadeiro se e somente se receber duas expressões que forem verdadeiras.
- or: retorna falso se e somente se receber duas expressões que forem falsas.
- not: retorna falso se receber uma expressão verdadeira e vice-versa.

Identidade:

- is: retorna verdadeiro se receber duas referências ao mesmo objeto e falso em caso contrário.

Filiação:

- in: retorna verdadeiro se receber um item e uma lista e o item ocorrer uma ou mais vezes na lista e falso em caso contrário.

Com operadores lógicos é possível construir condições mais complexas para controlar desvios condicionais e laços.

Exemplos

○ Exemplo usando o operador **and**

```
# Verifica se x é menor que y E y é menor que z  
if x < y and y < z:  
    print('x é menor que y e y é menor que z')  
else:  
    print("A condição não é verdadeira")
```

Exemplos

○ Exemplo usando o operador **or**

```
x = 5
```

```
y = 10
```

```
# Verifica se x é menor que 10 OU y é menor que 5
if x < 10 or y < 5:
    print("Pelo menos uma das condições é verdadeira")
else:
    print("Nenhuma das condições é verdadeira")
```

Exemplos

- Exemplo usando o operador **not**

x = 5

```
# Verifica se x NÃO é igual a 10
```

```
if not x == 10:
```

```
    print("x não é igual a 10")
```

```
else:
```

```
    print("x é igual a 10")
```

Exemplos

- O operador `in` é usado para verificar se um valor está presente em uma sequência, como uma lista, uma tupla, uma string, etc.

Exemplo usando o operador **in (está em)**

```
frutas = ["maçã", "banana", "laranja", "uva"]
```

```
# Verifica se "banana" está na lista de frutas
```

```
if "banana" in frutas:
```

```
    print("Sim, banana está na lista de frutas")
```

```
else:
```

```
    print("Não, banana não está na lista de frutas")
```

```
# Verifica se "morango" está na lista de frutas
```

```
if "morango" in frutas:
```

```
    print("Sim, morango está na lista de frutas")
```

```
else:
```

```
    print("Não, morango não está na lista de frutas")
```

Exemplos

- O operador `is` é usado para verificar se dois objetos têm o mesmo id, o que significa que eles ocupam o mesmo local na memória.

Exemplo usando o operador `is` (**aponta para o mesmo objeto**):

```
x = [1, 2, 3]
```

```
y = [1, 2, 3]
```

```
# Verifica se x e y são o mesmo objeto
```

```
if x is y:
```

```
    print("x e y são o mesmo objeto")
```

```
else:
```

```
    print("x e y não são o mesmo objeto")
```

```
# Altera y para referenciar o mesmo objeto que x
```

```
y = x
```

```
# Verifica se x e y são o mesmo objeto depois da alteração
```

```
if x is y:
```

```
    print("Agora x e y são o mesmo objeto")
```

```
else:
```

```
    print("x e y não são o mesmo objeto")
```

```
ESTRUTURA <expressao> :  
    procedimento1  
    procedimento2  
    ...
```

Estruturas de Controle de Fluxo

Estruturas Condicionais if-else-elif

if (se)

if..else (se..senão)

if..elif..else
(se..senão..senão
se)

```
x = int(input('digite um valor: '))
if x<10:
    print('x<10')
else:
    print('x>=10')
```

```
x = int(input('digite um valor: '))
if x>10:
    print('x maior que 10')
elif x<=10 and x>=5:
    print('x entre 5 e 10')
else:
    print('x menor que 5')
```

No Python, não há a estrutura switch.

Estruturas Condicionais if-else-elif

```
if <condição>:  
    <bloco de código>  
elif <condição>:  
    <bloco de código>  
elif <condição>:  
    <bloco de código>  
else:  
    <bloco de código>
```

O comando ELIF é uma abreviação de “else if” e é usado quando há mais de uma condição a ser testada. O ELIF permite que o programa teste várias condições e execute um bloco de código diferente para cada uma delas.

```
temp = int(raw_input('Entre com a temperatura: '))  
  
if temp < 0:  
    print 'Congelando...'<br/>elif 0 <= temp <= 20:  
    print 'Frio'<br/>elif 21 <= temp <= 25:  
    print 'Normal'<br/>elif 26 <= temp <= 35:  
    print 'Quente'<br/>else:  
    print 'Muito quente!'
```

Estruturas Condicionais

```
i = 0  
while i<10:  
    print(i)  
    i +=1
```

```
for i in [0, 'a', 2, 5, 1]:  
    print(i)
```

Quando os valores a serem atribuídos são números crescentes, pode-se usar o comando range(X), sendo assim, serão feitas X interações, atribuindo a variável valores de 0 a X-1.

```
for i in range(10):  
    print(i)
```

Estruturas de Repetição

```
>>> for i in range(0, 200000):
    print(i)
    if i == 5:
        break
```

```
>>> for x in range(0, 10):
    if x == 5:
        continue
    print(x)
```

Estruturas de Repetição

Laço com break:

```
frutas = ["maçã", "banana", "laranja", "uva"]
```

```
for fruta in frutas:
```

```
    if fruta == "laranja":
```

```
        break
```

```
print(fruta)
```

Resultado:

maçã

banana

Estruturas de Repetição

Laço com continue:

```
# Exemplo de um loop for com continue para pular a impressão de "laranja"
frutas = ["maçã", "banana", "laranja", "uva"]
for fruta in frutas:
    if fruta == "laranja":
        continue
    print(fruta)
```

Resultado:

maçã
Banana
uva

Estruturas de Repetição

Listas

Lista é um conjunto sequencial de valores, onde cada valor é identificado através de um índice.
O primeiro valor tem índice 0.

Uma lista em Python é declarada da seguinte forma:

```
Nome_Lista = [ valor1, valor2, ..., valorN]
```

Uma lista pode ter valores de qualquer tipo, incluindo outras listas.

Exemplo:

```
L = [3 , 'abacate' , 9.7 , [5 , 6 , 3] , "Python" , (3 , 'j')]
```

```
print(L[2])
```

9.7

```
print(L[3])
```

[5,6,3]

```
print(L[3][1])
```

6

Listas

Para alterar um elemento da lista, basta fazer uma atribuição de valor através do índice.
O valor existente será substituído pelo novo valor.

Exemplo: L[3]= 'morango'

```
print(L)
```

```
L = [3 , 'abacate' , 9.7 , 'morango', "Python" , (3 , 'j')]
```

Listas

A lista é uma estrutura mutável, ou seja, ela pode ser modificada. Na tabela a seguir estão algumas funções utilizadas para manipular listas.

Operações com Listas

Função	Descrição	Exemplo
len	retorna o tamanho da lista.	L = [1, 2, 3, 4] len(L) → 4
min	retorna o menor valor da lista.	L = [10, 40, 30, 20] min(L) → 10
max	retorna o maior valor da lista.	L = [10, 40, 30, 20] max(L) → 40
sum	retorna soma dos elementos da lista.	L = [10, 20, 30] sum(L) → 60
append	adiciona um novo valor na no final da lista.	L = [1, 2, 3] L.append(100) L → [1, 2, 3, 100]
extend	insere uma lista no final de outra lista.	L = [0, 1, 2] L.extend([3, 4, 5]) L → [0, 1, 2, 3, 4, 5]
del	remove um elemento da lista, dado seu índice.	L = [1,2,3,4] del L[1] L → [1, 3, 4]
in	verifica se um valor pertence à lista.	L = [1, 2 , 3, 4] 3 in L → True
sort()	ordena em ordem crescente	L = [3, 5, 2, 4, 1, 0] L.sort() L → [0, 1, 2, 3, 4, 5]
reverse()	inverte os elementos de uma lista.	L = [0, 1, 2, 3, 4, 5] L.reverse() L → [5, 4, 3, 2, 1, 0]

Mais operações em listas

- `clear()` Esvazia a lista:

```
>>> x = [1, 2, 3]
>>> x.clear()
>>> x
[]
```

- `pop()` Remove e retorna o elemento da posição especificada:

```
>>> x = ['a', 'b', 'c', 'd']
>>> x.pop(1)
'b'
>>> x
['a', 'c', 'd']
```

- `remove()` Remove o elemento especificado:

```
>>> x = ['a', 'b', 'c']
>>> x.remove('c')
>>> x
['a', 'b']
```

Operações com Listas

Concatenação (+)

```
a = [0, 1, 2]
b = [3, 4, 5]
c = a + b
print(c)
[0, 1, 2, 3, 4, 5]
```

Repetição (*)

```
L = [1, 2]
R = L * 4
print(R)
[1, 2, 1, 2, 1, 2, 1, 2]
```

Tipo	Operador	Descrição
Aritmético	+	Concatenação
Aritmético	*	Múltiplas concatenações
Relacional	==	Igual a
Relacional	!=	Diferente de
Relacional	<>	Diferente de
Relacional	>	Maior que
Relacional	>=	Maior ou igual a
Relacional	<	Menor que
Relacional	<=	Menor ou igual a

Operadores em Listas

Criação de Listas com range ()

A função range() define um intervalo de valores inteiros.

Associada a list(), cria uma lista com os valores do intervalo.

A função range() pode ter de 1 a 3 parâmetros:

- range(n) - gera um intervalo de 0 a n-1
- range(i , n) - gera um intervalo de i a n-1
- range(i , n, p) - gera um intervalo de i a n-1 com intervalo p entre os números

Exemplos:

```
L1 = list(range(5))
```

```
print(L1)
```

```
[0, 1, 2, 3, 4]
```

```
L2 = list(range(3,8))
```

```
print(L2)
```

```
[3, 4, 5, 6, 7]
```

```
L3 = list(range(2,11,3))
```

```
print(L3)
```

```
[2, 5, 8]
```

Tuplas

Tupla, assim como a Lista, é um conjunto sequencial de valores, onde cada valor é identificado através de um índice.

A principal diferença entre elas é que as tuplas são imutáveis, ou seja, seus elementos não podem ser alterados.

Dentre as utilidades das tuplas, destacam-se as operações de empacotamento e desempacotamento de valores.

Tuplas

Uma tupla em Python é declarada da seguinte forma:

```
Nome_tupla = (valor1, valor2, ..., valorN)
```

Exemplo:

```
T = (1,2,3,4,5)
```

```
print(T)
```

```
(1, 2, 3, 4, 5)
```

```
print(T[3])
```

```
4
```

Tipo	Operador	Descrição
Aritmético	+	Concatenação
Aritmético	*	Múltiplas concatenações
Relacional	==	Igual a
Relacional	!=	Diferente de
Relacional	<>	Diferente de
Relacional	>	Maior) que
Relacional	>=	Maior ou igual a
Relacional	<	Menor que
Relacional	<=	Menor) ou igual a

Operadores em Tuplas

```
>>> dir(tuple)
['__add__', '__class__', '__contains__', '__delattr__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__getitem__', '__getnewargs__',
 '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__iter__', '__le__', '__len__', '__lt__', '__mul__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rmul__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', 'count', 'index']
```

O comando `dir(tuple)` retorna os atributos e métodos das tuplas.

Operações com Tuplas

Resumindo.... Tupla X Lista

- Tupla, assim como a Lista, é um conjunto sequencial de valores, onde cada valor é identificado através de um índice. A principal diferença entre elas é que as tuplas são imutáveis, ou seja, seus elementos não podem ser alterados.
- Dentre as utilidades das tuplas, destacam-se as operações de empacotamento e desempacotamento de valores.
- Uma tupla em Python é declarada da seguinte forma:
`Nome_tupla = (valor1, valor2, ..., valorN)`
- **Tuplas**, assim como as listas, são posições da memória: **Empacotar** significa agrupar **valores**.

Resumindo.... Tupla X Lista

Uma ferramenta muito utilizada em tuplas é o desempacotamento, que permite atribuir os elementos armazenados em uma tupla a diversas variáveis.

Exemplo:

```
T = (10,20,30,40,50)
```

```
a,b,c,d,e = T
```

```
print("a=",a,"b=",b)
```

```
a= 10 b= 20
```

```
print("d+e=",d+e)
```

```
d+e= 90
```

Dicionários

Dicionário é um conjunto de valores, onde cada valor é associado a uma chave de acesso.

Um dicionário em Python é declarado da seguinte forma:

```
Nome_dicionario = { chave1 : valor1, chave2 : valor2, chave3 : valor3, ..... chaveN : valorN}
```

Exemplo: D={"arroz": 17.30, "feijão":12.50,"carne":23.90,"alface":3.40}

```
print(D)
```

```
{'arroz': 17.3, 'carne': 23.9, 'alface': 3.4, 'feijão': 12.5}
```

```
print(D["carne"])
```

```
23.9
```

```
>>> dir(dict)
[ '__class__', '__contains__', '__delattr__',
  '__delitem__', '__dir__', '__doc__', '__eq__',
  '__format__', '__ge__', '__getattribute__',
  '__getitem__', '__gt__', '__hash__', '__init__',
  '__init_subclass__', '__iter__', '__le__', '__len__',
  '__lt__', '__ne__', '__new__', '__reduce__',
  '__reduce_ex__', '__repr__', '__setattr__',
  '__setitem__', '__sizeof__', '__str__',
  '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get',
  'items', 'keys', 'pop', 'popitem', 'setdefault',
  'update', 'values']
```

O comando `dir(dict)` retorna os atributos e métodos dos dicionários.

Operações com Dicionários

Operações com dicionários

- `clear()` Esvazia o dicionário:

```
>>> d = {'nome': 'joao', 'idade': 20}  
>>> d.clear()  
>>> d  
{}
```

- `get()` Retorna o valor da chave passada como argumento. Caso haja um segundo argumento, será a resposta *default*, isto é, caso a chave não exista no dicionário, irá retornar o default.

```
>>> d = {'nome': 'joao', 'idade': 20}  
>>> d.get('nome')  
'joao'  
>>> d.get('teste')  
>>> d.get('teste', 'nao encontrado')  
'nao encontrado'
```

Strings

- As strings foram abordadas como tipos primitivos de dados, mas são também cadeias de caracteres.
- São sequências ordenadas, imutáveis e homogêneas.
- Ordenadas, pois cada elemento, isto é, cada letra possui seu índice, na ordem que estão armazenados. Imutáveis, pois não podem ser alteradas. Homogêneas, pois só aceitam um tipo de dado.

```
>>> P = 'Palavra'  
>>> P[0]  
'P'  
>>> P[-1]  
'a'  
>>> P[5]  
'r'
```

Em Python, os índices negativos permitem acessar elementos de uma sequência (como strings, listas, etc.) de trás para frente. **P[-1]** retorna o **último caractere** da string (neste caso, 'a').

Strings

Tipo	Operador	Descrição
Aritmético	+	Concatenação
Aritmético	*	Múltiplas concatenações
Relacional	==	Igual a
Relacional	!=	Diferente de
Relacional	<>	Diferente de
Relacional	>	Maior(alfabeticamente) que
Relacional	>=	Maior(alfabeticamente) ou igual a
Relacional	<	Menor(alfabeticamente) que
Relacional	<=	Menor(alfabeticamente) ou igual a

Operadores em Strings

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__getitem__', '__getnewargs__',
 '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__iter__', '__le__', '__len__', '__lt__', '__mod__',
 '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', 'capitalize', 'casefold', 'center',
 'count', 'encode', 'endswith', 'expandtabs', 'find',
 'format', 'format_map', 'index', 'isalnum', 'isalpha',
 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
 'islower', 'isnumeric', 'isprintable', 'isspace',
 'istitle', 'isupper', 'join', 'ljust', 'lower',
 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
 'split', 'splitlines', 'startswith', 'strip',
 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

O comando `dir(str)` retorna os atributos e métodos das listas.

Operações com Strings

- `capitalize()` Retorna a string com o primeiro caractere em maiúsculo e todos os caracteres em minúsculo:

```
>>> F = 'veja esta frase'  
>>> F.capitalize()  
'Veja esta frase'
```

- `count()` Retorna quantas ocorrências há do argumento passado:

```
>>> a = 'palavra'  
>>> a.count('a')  
3
```

- `index()` Retorna o índice da primeira ocorrência do argumento passado:

```
>>> a = 'palavra'  
>>> a.index('a')  
1
```

- `isalnum()` Retorna verdadeiro se todos os caracteres da frase forem alfanuméricos:

```
>>> A = 'abc11+'  
>>> B = 'abc11'  
>>> A.isalnum()  
False  
>>> B.isalnum()  
True
```

Operações com Strings

Operações com Strings

- isalpha() Retorna verdadeiro se todos os caracteres da frase forem alfabéticos:

```
>>> A = 'abc11'  
>>> B = 'abc'  
>>> A.isalpha()  
False  
>>> B.isalpha()  
True
```

- isdecimal() Retorna verdadeiro se todos os caracteres da frase representarem um número decimal:

```
>>> A = '10.9'  
>>> B = '5'  
>>> A.isdecimal()  
False  
>>> B.isdecimal()  
True
```

- islower() Retorna verdadeiro se todos os caracteres da frase forem minúsculos:

```
>>> A = 'abCD'  
>>> B = 'abcd'  
>>> A.islower()  
False  
>>> B.islower()  
True
```

- isnumeric() Retorna verdadeiro se todos os caracteres da frase forem numéricos:

```
>>> A = 'ab12'  
>>> B = '1234'  
>>> A.isnumeric()  
False  
>>> B.isnumeric()  
True
```

Operações com Strings

- `isupper()` Retorna verdadeiro se todos os caracteres da frase forem maiúsculos

```
>>> A = 'abCD'  
>>> B = 'ABCD'  
>>> A.isupper()  
False  
>>> B.isupper()  
True
```

- `lower()` Retorna a string com todos os caracteres em minúsculo:

```
>>> F = 'UMA FRASE'  
>>> F.lower()  
'uma frase'
```

- `replace()` Retorna o string trocando a primeira string do argumento pela segunda:

```
>>> F = 'Veja uma frase'  
>>> F.replace(' ', '-')  
'Veja-uma-frase'  
>>> F.replace('uma', 'alguma')  
'Veja alguma frase'
```

- `split()` Retorna a string separada de acordo com o separador, passado com argumento. Por padrão o separador é `'.'`:

```
>>> F = 'Veja uma frase'  
>>> F.split()  
['Veja', 'uma', 'frase']  
>>> F.split('a')  
['Vej', ' um', ' fr', 'se']
```

- `upper()` Retorna a string com todos os caracteres em maiúsculo:

```
>>> F = 'uma frase'  
>>> F.upper()  
'UMA FRASE'
```

```
>>> A = 'abc'; B = 'def'  
>>> A + B  
'abcdef'  
>>> A < B  
True
```

Strings

Comandos e Funções

- Funções auxiliam a automatizar procedimentos que são recorrentes no código, evitando a necessidade de escrevê-los várias vezes.
- Bem como as outras estruturas do Python, a estrutura de definição de funções também segue as regras da indentação e dos dois pontos.
- Para definir a função, usa-se a palavra reservada `def`, seguida do nome da função, dos parâmetros entre parênteses e dos dois pontos.
- Os procedimentos devem ficar abaixo desta linha e indentados.
- Para retornar valores, ou apenas para indicar o término da função, deve-se usar a palavra `return`.
- Exemplo: `minhaprimeirafunc.py`

Exemplo 1

```
def minhafuncsoma(a, b):  
    s = a + b  
    return s  
  
#programa principal  
x = 2  
y = 3  
z = minhafuncsoma(x, y)
```

```
def fatorial(num):  
  
    if num <= 1:  
        return 1  
    else:  
        return(num * fatorial(num - 1))
```

A função é recursiva.

Exemplo 2

Exemplo 3

```
def fibonacci(n):  
    a, b = 0, 1  
    for _ in range(n):  
        a, b = b, a + b  
    return a  
  
# Exemplo de uso:  
n = 10  
print(fibonacci(n)) # Saída: 55 (o 10º  
número na sequência de Fibonacci)
```

Exemplo 4

```
def maior(x, y):  
    if x>y:  
        print(x)  
    else:  
        print(y)
```

```
>>> maior(4,7)
```

7

Recursividade

```
>>>def factorial(n):
...     if n <= 1:
...         return 1
...     return n * factorial(n - 1)
```

A recursividade é um tipo de iteração (repetição) na qual uma função chama a si mesma repetidamente até que uma condição de saída seja satisfeita. Ao lado, temos um exemplo de uma função responsável por calcular o factorial de números positivos inteiros e demonstra como uma função pode chamar a ela mesma utilizando a propriedade recursiva.

Return

O comando return é usado para retornar um valor de uma função e encerrá-la. Caso não seja declarado um valor de retorno, a função retorna o valor None (que significa nada, sem valor atribuído).

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

#programa principal
s=soma(3, 5)
print("soma = ", s)
```

→ Resultado da execução:
soma = 8

Observações:

- a) O valor da variável total, calculado na função soma, retornou da função e foi atribuído à variável s.
- b) O comando após o return foi ignorado.

Valor Padrão

É possível definir um valor padrão para os parâmetros da função. Neste caso, quando o valor é omitido na chamada da função, a variável assume o valor padrão.

```
def calcula_juros(valor, taxa=10):  
    juros = valor*taxa/100  
    return juros
```

```
>>> calcula_juros(500)  
50.0
```

Variáveis Globais e Locais

- **Global:** é declarada no início de um programa, podendo ser usada por qualquer função subordinada ao programa principal. É visível a todas as funções subordinadas à função principal. Ocupa memória durante todo o ciclo do programa. Declarada uma única vez.
- **Local:** é declarada dentro de uma função e é somente válida dentro da função à qual está declarada. Demais funções não poderão fazer uso dela, pois não a visualizam.
 - Ganho em espaço em memória, torna o programa mais eficiente. Fica na memória enquanto executar a função.

Variáveis Globais e Locais em C

The screenshot shows the Dev-C++ IDE interface with the title bar "Untitled1 - Dev-C++ 5.11". The menu bar includes File, Edit, Search, View, Project, Execute, Tools, AStyle, Window, and Help. The toolbar contains various icons for file operations like Open, Save, Print, and Build. The status bar at the bottom shows "Line: 11 Col: 41 Sel: 0 Lines: 14 Length: 147 Insert Done parsing in 0,078 seconds". The taskbar at the bottom has icons for Windows, File Explorer, Internet Explorer, HP, Google Chrome, Microsoft Excel, Microsoft Word, Microsoft PowerPoint, and the Dev-C++ icon.

```
1 #include<stdio.h>
2
3
4 int a,b,c; // variaveis globais
5
6 int main()
7 {
8     int fatorial()
9     {
10        int fat=1, cont; // variaveis locais
11    }
12
13
14 }
```

```
def func():  
    y = 5 # Variável local  
    print(y) # Acessando a variável local  
  
func() # Saída: 5  
  
print(y) # Erro! 'y' não está definida fora da função
```

```
x = 10 # Variável global  
  
def func():  
    print(x) # Acessando a variável global dentro da função  
  
func() # Saída: 10
```

Variáveis locais e globais em Python

Alterando variáveis globais dentro de funções

Para modificar uma variável global dentro de uma função, é necessário usar a palavra-chave `global`. Isso informa ao Python que você quer usar a variável global e não criar uma nova variável local.

```
x = 10 # Variável global

def func():
    global x
    x = 20 # Modificando a variável global
    print(x)

func() # Saída: 20
print(x) # Saída: 20
```

```
x = 10 # Variável global

def func():
    x = 20 # Variável local (não modifica a global)
    print(x)

func() # Saída: 20
print(x) # Saída: 10 (a variável global permanece a mesma)
```

Alterando variáveis globais dentro de funções

Sem a palavra-chave `global`, ao tentar atribuir um valor a uma variável que tem o mesmo nome que uma global dentro de uma função, o Python criará uma nova variável local, e a global permanecerá inalterada.

Resumindo...

- Variável Global: Definida fora de funções, acessível em qualquer parte do código.
- Variável Local: Definida dentro de funções, acessível apenas dentro dessa função.
- *global Keyword*: Necessária para modificar variáveis globais dentro de funções.



Orientação a Objetos

- É importante ressaltar que os tipos de dados tratados são, na verdade, classes, sendo elas um conceito primordial de Orientação a Objetos.
- Em SPOLOGP será uma abordagem inicial, pois o conteúdo será melhor desenvolvido no 2º ano do curso.
- Os chamados tipos de dados de Python (int, float, str, list, etc), agora são formalmente apresentados como classes. As classes facilitam a modularização e abstração, são elas que fazem do Python uma linguagem orientada a objetos.

Classe

- uma classe é uma estrutura fundamental que define um molde ou modelo para criar objetos (instâncias). Ela agrupa dados e comportamentos que são comuns a um tipo específico de objeto, permitindo organizar o código de maneira mais modular, reutilizável e fácil de manter.
- Uma classe agrupa dados (chamados de **atributos**) e funções (chamadas de **métodos**) que operam sobre esses dados. Em outras palavras, uma classe descreve como os objetos devem ser estruturados e como eles devem se comportar.

Características principais de uma classe

- **Encapsulamento:** A classe permite encapsular (agrupar) dados e comportamentos, protegendo seus atributos para que não possam ser acessados ou alterados diretamente.
- **Herança:** Em Python, uma classe pode herdar atributos e métodos de outra classe, promovendo reutilização de código.
- **Polimorfismo:** Diferentes classes podem definir métodos com o mesmo nome, mas comportamentos distintos.

```
class NomeDaClasse:  
    # Construtor da classe, inicializa os atributos  
    def __init__(self, atributo1, atributo2):  
        self.atributo1 = atributo1  
        self.atributo2 = atributo2  
  
    # Um método da classe  
    def metodo(self):  
        print(f"O valor do atributo1 é {self.atributo1}")
```

O parâmetro `self` deve ser declarado explicitamente na definição do método, pois servirá para acessar os próprios métodos e atributos. É importante ressaltar que `self` não é uma palavra reservada em Python, mas é amplamente usado por convenção e boas práticas de programação.

Classe em Python – Exemplo 1

Objeto

Um objeto é definido como uma estrutura de dados que é uma instância de uma classe.

Ele é criado a partir de um molde, e este molde é chamado de classe. Pode-se fazer analogia com tipos de dados, por exemplo:

Uma variável a é do tipo inteira; um objeto a é da classe int.

Classes são como tipos de dados, mas mais incorporados, e os objetos são as variáveis dessas classes.

Objeto

Classe é definida como um agrupamento de valores e de operações.

Frequentemente classes diferentes possuem características comuns. As classes diferentes podem compartilhar valores comuns e podem executar as mesmas operações. Em Python tais relacionamentos são expressados usando derivação e herança.

Quando uma classe é utilizada para criar uma instância, chamamos essa instância de objeto. Cada objeto tem seus próprios valores de atributos, mas compartilha a estrutura e os métodos definidos pela classe.

Por exemplo, você pode criar dois objetos da classe Carro: um com a cor vermelha e outro azul.

Exemplo 2

Existindo uma classe definida por um usuário `MinhaClasseCaderno`, então um objeto `Caderno1` pode ser criado da seguinte forma:

```
>>> Caderno1 = MinhaClasseCaderno ()
```

Atributo

- Também chamados de propriedades ou variáveis de instância, atributos são os dados que a classe armazena. Por exemplo, uma classe Carro poderia ter atributos como cor, marca, modelo, e velocidade.

```
>>> class Caderno():
    cor = 'preto'
    np = 10

>>> NovoCaderno = Caderno()
>>> NovoCaderno.cor
'preto'
```

Exemplo 3

Os atributos são os valores que existem dentro do objeto.

Método

- São as funções ou comportamentos que podem ser executados pelos objetos criados a partir da classe. Em nosso exemplo de Carro, métodos poderiam ser acelerar(), frear() e virar().

Construtor

O construtor é um método especial da classe, responsável por inicializar o objeto quando ele é criado. Na maioria das linguagens, ele é chamado automaticamente quando o objeto é instanciado.

O método construtor é um método especial das classes. Ele é executado sempre que uma nova classe é iniciada e é denotado por `__init__`.

```
class Carro:  
    def __init__(self, marca, cor):  
        self.marca = marca # Atributo  
        self.cor = cor      # Atributo  
  
    def acelerar(self):      # Método  
        print(f"O {self.marca} está acelerando!")  
  
# Criando um objeto da classe Carro  
meu_carro = Carro("Toyota", "Vermelho")  
meu_carro.acelerar() # Chamando o método acelerar
```

Neste exemplo: A classe Carro tem dois atributos: marca e cor. O método acelerar() define um comportamento para os objetos da classe. O objeto meu_carro é uma instância da classe Carro, e ele pode chamar o método acelerar().

Exemplo 4 em Python

```
class Pessoa:  
    # Construtor da classe  
    def __init__(self, nome, idade):  
        self.nome = nome  
        self.idade = idade  
  
    # Método que imprime informações da pessoa  
    def saudacao(self):  
        print(f"Olá, meu nome é {self.nome} e tenho {self.idade} anos.")  
  
# Criando um objeto (instância) da classe Pessoa  
pessoa1 = Pessoa("Maria", 30)  
  
# Chamando o método da classe  
pessoa1.saudacao() # Saída: Olá, meu nome é Maria e tenho 30 anos.
```

Exemplo 5 em Python

Acessor e Modificador

- São chamados de métodos acessores os métodos que acessam um atributo do objeto, mas não o modificam. Enquanto que os métodos modificadores acessam e modificam. Esses métodos também são chamados de getters e setters, respectivamente.

```
>>> class Caderno():
    cor = 'preto'
    np = 10
    def getCor(self):
        return self.cor
    def getNp(self):
        return self.np
    def setCor(self, novaCor):
        self.cor = novaCor
    def setNp(self, novoNp):
        self.np = novoNp
```

```
>>> C = Caderno()
>>> C.getCor()
'preto'
>>> C.setCor('amarelo')
>>> C.getCor()
'amarelo'
```

Acessor e Modificador

No exemplo acima, observa-se que existem dois métodos acessores, `getCor` e `getNp` e dois métodos modificadores `setCor` e `setNp`. No exemplo, também é possível observar o uso do primeiro parâmetro, o `self`.

```
>>> C = Caderno()  
>>> C.setNp(10)  
>>> C.np  
10  
>>> C.delNp()  
>>> C.np  
Traceback (most recent call last):  
  File "<pyshell#143>", line 1, in <module>  
    C.np  
AttributeError: 'Caderno' object has no attribute 'np'
```

```
>>> class Caderno():  
    def getCor(self):  
        return self.cor  
    def getNp(self):  
        return self.np  
    def setCor(self, novaCor):  
        self.cor = novaCor  
    def setNp(self, novoNp):  
        self.np = novoNp  
    def delNp(self):  
        del self.np
```

o método deletor, ou deleter, é o método capaz de deletar um atributo de uma classe.

Delete ou Deletor

Modificadores de acesso

Outro conceito importante na orientação a objetos é o acesso. Isto é, a permissão de acessar atributos e métodos das classes. Geralmente, em linguagens clássicas de POO, usam-se três palavras reservadas para definir diferentes tipos de acesso. Elas são chamadas de modificadoras de acesso:

- private, privado. O acesso privado permite que os atributos e classes sejam acessados apenas dentro da própria classe.
- public, público. Os membros públicos podem ser acessados em qualquer lugar, até mesmo fora da classe.
- protected, protegido. Os membros protegidos podem ser acessados apenas dentro da classe e de suas sub-classes.

Modificadores de acesso

Em Python, não há estruturas e mecanismos que restrinjam o acesso. Há apenas uma

convenção ao se dar um prefixo ao nome do método ou atributo com um ou dois caracteres: underscore "_".

Por padrão, os atributos e métodos em Python são públicos.

Os prefixados com um caractere underscore são protegidos e os com dois privados.

De fato, apenas o privado não pode ser acessado diretamente, mas se a classe for instanciada em um objeto, poderá ser acessado usando-se a sintaxe: `objeto._class_privado`.

Herança

- A herança em Python é um dos princípios fundamentais da programação orientada a objetos (POO). Ela permite que uma classe (denominada classe filha ou subclasse) herde atributos e métodos de outra classe (denominada classe pai ou superclasse). Isso promove reutilização de código e ajuda a organizar e modelar os dados de maneira eficiente.

Herança

Python suporta tipos de herança como:

- **Herança simples**: Uma classe filha herda de uma única classe pai.
- **Herança múltipla**: Uma classe filha pode herdar de várias classes pais.

Exemplo de Herança

```
# Definindo a classe pai
class Animal:
    def __init__(self, nome):
        self.nome = nome

    def som(self):
        return "Som de animal"

# Definindo a classe filha, que herda de Animal
class Cachorro(Animal):
    def som(self):
        return "Latido"

# Definindo outra classe filha
class Gato(Animal):
    def som(self):
        return "Miau"

# Criando instâncias
cachorro = Cachorro("Rex")
gato = Gato("Felix")

# Acessando métodos e atributos
print(cachorro.nome) # Rex
print(cachorro.som()) # Latido
print(gato.nome) # Felix
print(gato.som()) # Miau
```

Classe Pai (`Animal`): Define um método genérico `som` e um atributo `nome`.

Classe Filha (`Cachorro` e `Gato`): Herdam o atributo `nome` da classe pai, mas sobrescrevem o método `som`, fornecendo um comportamento específico para cada animal.

Vantagens da Herança

- O Reutilização de Código: Você pode evitar duplicação de código usando herança.
- O Modularidade: As classes podem ser modificadas ou estendidas sem mudar o código original.
- O Facilidade de manutenção: Mudanças na classe pai podem beneficiar todas as subclasses, centralizando ajustes.

Vantagens de se utilizar POO

- Reutilização de código: Você pode criar múltiplos objetos com base na mesma classe.
- Encapsulamento: Esconde detalhes internos dos objetos e expõe apenas o necessário.
- Modularidade: Organiza o código em partes menores e mais manejáveis.
- Herança e polimorfismo: Permitem que novas classes herdem comportamentos e dados de outras, facilitando a extensão e modificação do código sem quebrar funcionalidades existentes.

A classe, portanto, é a base da orientação a objetos e é usada para modelar objetos do mundo real ou abstrações no software.