**Introdução ao Python**

Python nasceu em 1989 como um hobby, do programador GUIDO VAN ROSSUM. A inicial era dar continuidade a linguagem ABC.

- Uma linguagem fácil e intuitiva

- Código aberto, para que todos possam contribuir

- Código tão inteligível quanto inglês

- Adequada para tarefas diárias, e produtiva!

A primeira versão em Fev/1991 Versão pública 0.9.0.

Em 1994 lança a versão 1.0

Em 1995 lança a versão 1.2

Em 2000 lança a versão 2.0 e nasce a List Comprehension (Melhoria no coletor de lixo)

Em 2001 nasce a versão 2.1 Python Software Foundation (PSF) a partir daqui todo o código, documentação e especificações da linguagem.

Em 2008 é lançada a versão 3.0

Atualmente a versão é a 3.10.2.

Python não é boa para APP Mobile

Linguagem muito versátil

- Tipagem dinâmica e forte

- Multiplataforma e multiparadigma

- Comunidade gigante e ativa

- Curva de aprendizado baixa

**Tipos de Dados**

|  |  |
| --- | --- |
| Texto | str |
| Numérico | Int, float, complex |
| Sequência | List, tuple, range |
| Mapa | Dict |
| Coleção | Set, fronzenset |
| Booleano | bool |
| Binário | Bytes, bytearray, memoryview |
|  |  |

**Modo interativo**

O interpretador Python pode executar em modo que possibilite o desenvolvedor a escrever código, e ver o resultado na hora.

Existem duas formas de iniciar chamando o interpretador(python) no terminal ou executando o script com a flag -i (python -i app.py) também no terminal.

**Função DIR**

Sem argumentos, retorna lista de nomes no escopo local atual.

Com argumentos, retorna uma lista de atributos validos para o objeto

Exemplo

dir()

dir(100)

**Função HELP**

Invoca o sistema de ajuda integrado. É possível fazer buscar em modo interativo ou informar por parâmetro qual o nome do módulo, função, classe, método ou variável

Exemplo

help()

help(100)

**Variáveis**

São valores que podem sofrer alterações no decorrer da execução do programa.

**Constantes**

Uma constante nasce com o mesmo valor e não altera na execução do programa, ou seja, não altera o valor até o final da execução do programa.

Porém em python não tem constantes, mas que isso ocorra fazemos uma convenção que o programador usa cria a variável em maiúscula.

Exemplo

STATES = [ ‘SP’, ‘RJ’, ’MG’]

AMOUNT = 30.2

**BOAS PRÁTICAS**

* O padrão de nomes deve ser *snake case*. (EX:taxaJuros, valor\_total, etc...).
* Escolher nomes sugestivos.
* Nome de constantes todos em maiúsculo. (EX:TAXAJUROS).

**CONVERSÃO DE TIPOS**

Em alguns momentos é necessário ou será necessário converter o tipo de uma variável para manipular de forma diferente. Por exemplo:

Variáveis do tipo *String,* que armazenam números e precisamos fazer alguma operação matemática com esse valor.

Inteiro para float

preco = 10

print(preco) >>> 10

preco = float(preco)

print(preco)

>>> 10.0

preco = 10 / 2

print(preco)

>>> 5.0

Float para inteiro

preco = 10.30

print(preco)

>>> 10.3

preco = int(preco)

print(preco)

>>> 10

Conversão por divisão

preco = 10

print(preco)

>>> 10

print(preco / 2)

>>> 5.0

print(preco // 2)

>>> 5

Numérico para string

preco = 10.50

idade = 28

print(str(preco))

>>> 10.5

print(str(idade))

>>> 28

texto = f”idade {idade} preco{preco}”

print(texto)

>>> idade 28 preco 10.5

String para número

preco = “10.50”

idade = “28”

print(float(preco))

>>> 10.5

print(int(idade))

>>> 28

Erro de conversão

Texto

Descrição gerada automaticamente

**FUNÇÃO INPUT(ENTRADA)**

A função builtin *input* é utilizada quando queremos ler dados da entrada padrão (teclado). Ela recebe um argumento de tipo string, que é exibido para o usuário na saída padrão(tela). A função lê a entrada, converte para string e retorna o valor.

Exemplo

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

**FUNÇÃO INPUT(ENTRADA)**

A função builtin *print* é utilizada quando queremos exibir dados na saída padrão (tela). Ela recebe um argumento obrigatório do tipo varargs de objetos e 4 arqgumentos opcionais (sep, end, file e flush). Todos os objetos são convertidos para string, separados por *sep* e terminados por *end*. A string final é exibida para o usuário.

Exemplo

Texto

Descrição gerada automaticamente

**TIPOS DE OPERADORES**

**OPERADORES ARITMETICOS**

São os operadores executam operação matemáticas como adição, subtração, multiplicação e divisão.

print(1 + 1) >>> 2

print(2 - 1) >>> 1

print(2 \* 2) >>> 4

print(12 / 3) >>> 4.0

print(12 // 2) >>> 6 (Divisão Inteira)

print(10 % 3) >>> 1(Módulo) resto da divisão

print(2 \*\* 3) >>> 8(Exponenciação) 2^3 = 2\*2\*2 = 8

**PRECENDENCIAS MATEMATICAS**

print(10 – 5 \* 2) >>> 0

print((10 – 5) \* 2 ) >>> 10

print(10 \*\* 2 \* 2) >>> 200

print(10 \*\* (2 \* 2)) >>> 1000

print(10 / 2 \* 4) >>> 20.0

**OPERADORES DE COMPARAÇÃO**

Serve para comparar dois valores

saldo = 450

saque = 200

print(saldo == saque)

>>> false

saldo = 450

saque = 200

print(saldo != saque)

>>> true

saldo = 450

saque = 200

print(saldo > saque)

>>> true

print(saldo >= saque)

>>> true

saldo = 450

saque = 200

print(saldo < saque)

>>> false

print(saldo <= saque)

>>> false

|  |  |  |
| --- | --- | --- |
| Operador | Definição | Exemplo |
| == | Igualdade | 1 == 1 |
| != | Diferente | 2 != 3 |
| < | Menor que | 3 < 5 |
| <= | Menor ou Igual | 3 <= 5 |
| > | Maior que | 5 > 3 |
| >= | Maior ou igual | 5 >= 3 |

**OPERADORES DE ATRIBUIÇÃO**

Utilizados para definir o valor inicial ou sobrescrever o valor de uma variável.

Atribuição simples(=)

saldo = 500

print(saldo) >>> 500

Atribuição com adição

saldo = 500

saldo += 200

print(saldo) >>> 700

Atribuição com subtração

saldo = 500

saldo -= 200

print(saldo) >>> 300

Atribuição com divisão

saldo = 500

saldo /= 5

print(saldo) >>> 100.0

saldo = 500

saldo //= 5

print(saldo) >>> 100

Atribuição com módulo(resto)

saldo = 500

saldo %= 5

print(saldo) >>> 20

Atribuição com exponenciação

saldo = 80

saldo \*\*= 2

print(saldo) >>> 6400

**OPERADORES LOGICOS**

Utilizados em conjuntos com os operadores de comparação, para montar uma expressão logica.

O resultado retornado é um booleano.

Op\_comparacao + op\_logico + op\_comparacao... N ...

Operador E(and)

saldo = 1000

saque = 200

limite = 100

saldo >= saque *and* saque <= limite

>>> false

Operador OU(or)

saldo = 1000

saque = 200

limite = 100

saldo >= saque *or* saque <= limite

>>> true

Operador de Negação

Contatos\_emergencia = []

not 1000 > 1500

>>> true

not contatos\_emergencia

>>> true

not “saque 1500;”

>>> false

not “”

>>> true

|  |  |  |  |
| --- | --- | --- | --- |
| Operador | Representação | exemplo | Retorno |
| E | and | 2 >= 1 and 2 <= 1 | false |
| OU | or | 2 >= 1 and 2 <= 1 | true |
| not | not | not 1000 > 1500 | true |

**OPERADORES DE IDENTIDADE**

Utilizados para comparar se dois objetos testados ocupam a mesma posição na memória.

curso = “Curso de Python”

nome\_curso = curso

saldo, limite = 200, 200

curso is nome\_curso

>>> True

curso is not nome\_curso

>>> False

saldo is limite

>>> True

|  |  |
| --- | --- |
| Operador | Comparar |
| is | é |
| Is not | Não é |

**OPERADORES DE ASSOCIAÇÃO**

Utilizados para verificar se um objeto está presente em uma sequência.

curso = “Curso de Python”

frutas = [“laranja”, “uva”, “limão”]

saques = [1500, 100]

“Python” in curso

>>> True

“maça” not in frutas

>>> True

200 in saques

>>> False

|  |  |
| --- | --- |
| Operador | Comparar |
| in | esta |
| not in | Não esta |

**A ESTÉTICA**

Identar código é uma forma de manter o código fonte mais legível e manutenível. Mas em Python ela exerce um segundo papel, através da indentação o interpretador consegue determinar onde um bloco de comando inicia e onde ele termina.

Bloco de comando

As linguagens de programação costumam utilizar caracteres ou palavras reservadas para terminar o início e fim do bloco. Em java e C por exemplo, utilizamos chaves:

def sacar(self, valor: float) → None: # Inicio do bloco do método

if self.saldo >= valor: # inicio do bloco do if

self.saldo -= valor

# fim do bloco do if

# fim do bloco do método

**ESTRUTURAS CONDICIONAIS**

Permite o desvio de fluxo de controle. Quando determinadas expressões logicas são atendidas.

IF

Parar criar uma estrutura condicional simples, composta por um único desvio, podemos utilizar a palavra reservada if. O comando irá testar a expressão logica, e em caso de retorno verdadeiro as ações presentes no bloco de código do if serão executadas.

saldo = 2000.0

saque = float(input(“Informe o valor do saque: ”))

If saldo > = saque:

print(“Saque Realizado!”)

If saldo <= saque:

print(“Saldo Insulficiente!”)

IF / ELSE

Para criar uma estrutura condicional com dois desvios, podemois utilizar as palavras reservadas if e else. Como sabemos se a expressão lógica testada no if for verdadeira, então o bloco de código do if será executado. Caso contrário o bloco de código do else será executado.

saldo = 2000.0

saque = float(input(“Informe o valor do saque: ”))

If saldo > = saque:

print(“Saque Realizado!”)

else:

print(“Saldo Insulficiente!”)

IF / ELIF / ELSE

Em alguns cenários queremos mais de dois desvios, para isso podemos utilizar a palavra reservada elif. O elif é composto por uma nova expressão logica, que será testada e caso retorne verdadeiro o bloco de código do elif será executado. Não existe um número máximo de elifs que podemos utilizar, porém evite criar grandes estruturas condicionais, pois elas aumentam a complexidade do código.

opção = int(input(“Informe uma opção: [1] Sacar \n [2] Extrato: ”))

if opção == 1:

Valor=float(input(“Informe o valor a sacar: “))

elif opção == 2:

print(“Exibindo o extrato...”)

Else:

Sys.exit(“Opção invalida”)

IF ANINHADO

Podemos criar estruturas condicionais aninhadas, para isso basta adicionar estruturas if/elif/else dentro do bloco de código de estruturas if/elif/else

if conta\_normal:

if saldo >= saque:

print("Saque realizado com sucesso!")

elif saque <= (saldo + cheque\_especial):

print("Saque realizado com uso do cheque especial!")

elif conta\_universitaria:

if saldo >= saque:

print("Saque realizado com sucesso!")

else:

print("Saldo insulficiente!")

IF TERNÁRIO

O if ternário permite escrever uma condição em uma única linha. Ele é composto por três partes, a primeira parte é o retorno caso a expressão retorne verdadeiro, a segunda parte é a expressão logica e a terceira parte é o retorno caso a expressão não atendida.

status = “Sucesso” if saldo >= saque else “falha”

Print(f”{status} ao realizar o saque!”)

**ESTRUTURAS DE REPETIÇÃO**

Utilizadas para repetir um trecho de código um determinado número de vezes. Esse número pode ser conhecido previamente ou determinado através de uma expressão logica.

**FOR**

É usado para percorrer um objeto iterável. Faz sentido usar for quando sabemos o número exato de vezes que nosso bloco de código deve ser executado, ou quando queremos percorrer um objeto iterável.

texto = input(“Informe um texto: ”)

VOGAIS = “AEIOU”

for letra in texto:

if letra.upper() in VOGAIS:

print(letra, end=””)

print()

FOR/ELSE

texto = input(“Informe um texto: ”)

VOGAIS = “AEIOU”

for letra in texto:

if letra.upper() in VOGAIS:

print(letra, end=””)

else:

print()

FUNÇÃO RANGE

É uma função built-in do Python, ela é usada para produzir uma sequência de números inteiros a partir de um início(inclusivo) para um fim(exclusivo). Se usarmos range(i,j) sera produzido:

I, i+1, i+2, i+3, ..., j-1.

Ela recebe 3 argumentos: stop(obrigatório), start(opcional) e step opcional.

# range(stop) -> range object

# range(start, stop[, step]) -> range object

list(range(4))

>>> [0, 1, 2, 3]

utilizando range com for

for numero in range(0, 11):

print(numero, end=” “)

>>> 0 1 2 3 4 5 6 7 8 9 10

# exibindo a tabuada do 5

for numero in range(0, 51, 5):

print(numero, end=” “)

>>> 0 5 10 15 20 25 30 35 40 45 50

COMANDO WHILE

É usado para repetir um bloco de código várias vezes. Faz sentido usar while quando não sabemos o número exato de vezes que nosso bloco de código deve ser executado.

opção = -1

while opcao!= 0:

opcao = int(input(“[1] Sacar \n [2] Extrato \n [0] sair \n: ”))

if opção == 1:

print(“Sacado ....”)

elif opção == 2:

print(“Exibindo o extrato ....”)

else:

print(“Obrigado e volte sempre ....”)

CLASSE STRING

É famosa por ser rica em métodos e possuir uma interface muito fácil de trabalhar.

Em algumas linguagens manipular sequencias de caracteres não é um trabalho trivial, porém, em Python esse trabalho é muito simples.

nome = "CaRlos LOpeS"

print(nome.upper()) >>> CARLOS LOPES

print(nome.lower()) >>> carlos lopes

print(nome.title()) >>> Carlos Lopes

print()

#Remover Espaços

texto = " Ola Mundo "

print(texto.strip()) >>> Ola Mundo

print(texto.rstrip()) >>> Ola Mundo

print(texto.lstrip()) >>> Ola Mundo

print()

#Centralizar

texto2 = "Meu Curso de Python"

print(texto2.center(25)) >>> Meu Curso de Python

print(texto2.center(25, "#")) >>> ###Meu Curso de Python###

print(".".join(texto2)) >>> M.e.u. .C.u.r.s.o. .d.e. .P.y.t.h.o.n

INTERPOLAÇÃO DE VARIAVEIS

Temos 3 formas de interpolar variáveis em strings, a primeira é usando o sinal %, a segunda é utilizada o método format e a última é utilizando f string.

A primeira forma não é atualmente recomendada e seu uso em Python 3 é raro, por esse motivo iremos focar nas 2 últimas.

(%)

nome = “Carlos”

idade = 28

profissao = “Programador”

linguagem = “Python”

Print (“Olá, me chamo %s. Eu tenho %d anos de idade, trabalho como %s e estou matriculado no curso de %s.” % (nome, idade, profissao, linguagem))

>>> Olá, me chamo Carlos. Eu tenho 32 anos de idade, trabalho como Programador e estou matriculado no curso de Python.

Método format

nome = “Carlos”

idade = 28

profissao = “Programador”

linguagem = “Python”

Print (“Olá, me chamo {}. Eu tenho {} anos de idade, trabalho como {} e estou matriculado no curso de {}.” .format (nome, idade, profissao, linguagem))

Print (“Olá, me chamo {3}. Eu tenho {2} anos de idade, trabalho como {1} e estou matriculado no curso de {0}.” .format (linguagem, profissao, idade, nome))

Print (“Olá, me chamo {nome}. Eu tenho {idade} anos de idade, trabalho como {profissao} e estou matriculado no curso de {linguagem}.” .format (nome=nome, idade=idade, profissão=profissao, linguagem=linguagem))

Print (“Olá, me chamo {nome}. Eu tenho {idade} anos de idade, trabalho como {profissao} e estou matriculado no curso de {linguagem}.” .format(\*\*pessoa) )

>>> Olá, me chamo Carlos. Eu tenho 32 anos de idade, trabalho como Programador e estou matriculado no curso de Python.

F-STRING

nome = “Carlos”

idade = 28

profissao = “Programador”

linguagem = “Python”

Print (f“Olá, me chamo {nome}. Eu tenho {idade} anos de idade, trabalho como {profissao} e estou matriculado no curso de {linguagem}.”)

>>> Olá, me chamo Carlos. Eu tenho 32 anos de idade, trabalho como Programador e estou matriculado no curso de Python.

PI = 3.14159

Print(f”Valor de PI: {PI: .2f}”)

>>> Valor de PI: 3.14

Print(f”Valor de PI: {PI: 10.2f}”)

>>> Valor de PI: 3.14

FATIAMENTO

É uma técnica utilizada para retornar substrings (partes da string original), informando inicio (start), fim (stop) e passo (step): [start: stop[, step]].

nome = “Guilherme Arthur de Carvalho”

nome[0]

>>> “G”

nome[:9]

>>> “Guilherme”

nome[10:]

>>> “Arthur de Carvalho”

nome[10:16]

>>> “Arthur”

nome[10:16:2]

>>> “Atu”

nome[:]

>>> “Guilherme Arthur de Carvalho”

nome[:: -1]

>>> “ohlavraC ed ruhtrA emrehliuG”

STRING MULTIPLAS LINHAS

São definidas informando 3 aspas simples ou duplas durante a atribuição. Elas podem ocupar várias linhas do código, e todos os espaços em brancos são incluídos na string final.

nome = “guilherme”

mensagem = f””” Olá meu nome é {nome}, Eu estou aprendendo Python“””

>>>

Olá meu nome é Guilherme,

Eu estou aprendendo Python

nome = “guilherme”

mensagem = f’’’

Olá meu nome é {nome},

Eu estou aprendendo Python.

Essa mensagem tem diferentes recuos.

‘’’

>>>

Olá meu nome é {nome},

Eu estou aprendendo Python.

Essa mensagem tem diferentes recuos.

LISTAS

Podem armazenar de maneira sequencial qualquer tipo de objeto. Podemos criar listas utilizando o construtor *list,* a função range ou colocando valores separados por virgula dentro de colchetes. Listas são objetos mutáveis, portanto podemos alterar seus valores após a criação.

Exemplo

frutas = [“laranja”, “maca”, “uva”]

frutas = []

letras = list(“python”)

números = list(range(10))

carro = [“Ferrari”, “F8”, 4200000, 2020, 2900, “São Paulo”, True]

ACESSO DIRETO

A Lista é sequência, portanto podemos acessar seus dados utilizando índices. Contamos o índice de determinada sequência a partir do zero.

frutas = [“laranja”, “maca”, “uva”]

frutas[1] #maça

frutas[2] #uva

frutas = [“laranja”, “maca”, “uva”]

frutas[-1] #uva

frutas[-3] #laranja

LISTA ANINHADAS

Listas podem armazenar todos os tipos de objetos Payton, portanto podemos ter listas que armazenam outras listas. Com isso podemos criar estruturas bidimensionais (Tabelas), e acessar informando os índices de linha e coluna.

matriz = [

[1, “a”, 2],

[“b”, 3, 4],

[6, 5, “c”]

]

matriz[0] # [1, “a”, 2]

matriz[0][0] # 1

matriz[0][-1] #2

matriz[-1][-1] #c

LISTA FATIAMENTO

lista = [“P”, “y”, “t”, “h”, “o”, “n”]

lista[2:] #[“t”, “h”, “o”, “n”]]

lista[:2] #[“P”, “y”]

lista[1:3] #[“y”, “t”]

lista[0:3:2] # [“p”, “t”]

lista[::] #[“P”, “y”, “t”, “h”, “o”, “n”]

lista[::-1] #[“n”, “o”, “h”, “t”, “y”, “p”]

METODOS DA CLASSE LIST

Para adicionar valores *[].append*

lista = []

lista.append(1)

lista.append(“Python”)

lista.append([40, 30, 20])

print(lista) # [1, “Python”, [40, 30, 20]]

Para limpar valores *[].clear*

lista = [1, “Python”, [40, 30, 20]]

print(lista) # [1, “Python”, [40, 30, 20]]

lista.clear()

print(lista) #[]

Para copiar valores *[].copy*

lista = [1, “Python”, [40, 30, 20]]

lista2 = lista.copy()

print(lista2) # [1, “Python”, [40, 30, 20]]

Para contar valores *[].count*

cores = [“vermelho”, “azul”, “verde”, “azul”]

cores.count(“vermelho”) # 1

cores.count(“azul”) #2

cores.count(“verde”) #1

Para juntar valores *[].extend*

linguagens = [“python, “js”, “c”]

print(linguagens) # [“python, “js”, “c”]

linguagens.extend([“java”, “csharp”])

print(linguagens) # [“python, “js”, “c”, “java”, “csharp”]

Para pegar primeira posição valores na lista *[].index*

linguagens = [“python, “js”, “c”, “java”, “csharp”]

linguagens.index(“java”) # 3

linguagens.index(“python”) # 0

Para remover valores *[].pop*

Linguagens = [“python, “js”, “c”, “java”, “csharp”]

linguagens.pop() #csharp

linguagens.pop() #java

linguagens.pop() #c

linguagens.pop(0) #python

Para remover valores *[].remove*

linguagens = [“python, “js”, “c”, “java”, “csharp”]

linguagens.remove(“c”)

print(linguagens) #[“python, “js”, “java”, “csharp”]

Para inverter a ordem valores *[].reverse*

linguagens = [“python, “js”, “c”, “java”, “csharp”]

linguagens.reverse()

print(linguagens) #[“csharp”, “java”, “c”, “js”, “python”]

Para ordena valores *[].sort*

linguagens = [“python, “js”, “c”, “java”, “csharp”]

linguagens.sort() # [“c”, “csharp”, “java”, “js”, “python”]

linguagens = [“python, “js”, “c”, “java”, “csharp”]

linguagens.sort(reverse=true) # [“python, “js”, “java”, “csharp”, “c”]

linguagens = [“python, “js”, “c”, “java”, “csharp”]

linguagens.sort(key=lambda x: len(x)) # [“c” , “js”, “java”, “python”, “csharp”,]

linguagens = [“python, “js”, “c”, “java”, “csharp”]

linguagens.sort(key=lambda x: len(x), reverse=true) # [“python”, “csharp”, “java”, “js”, “c”]

Para verificar tamanho *len*

linguagens = ["python", "js", "c", "java", "csharp"]

print(linguagens) >>> 6

print(len(linguagens[1])) >>> 2

Para ordenar inteiráveis *sorted*

linguagens = ["python", "js", "c", "java", "csharp"]

print(sorted(linguagens, *key*=*lambda* *x*: len(x)))

>>> ['c', 'js', 'java', 'python', 'csharp']

print(sorted(linguagens, *key*=*lambda* *x*: len(x), *reverse*=True))

>>> ['python', 'csharp', 'java', 'js', 'c']

TUPLAS

São estruturas de dados muito parecidas com a lista, a principal diferença é que tuplas são imutáveis enquanto listas são mutáveis. Podemos criar tuplas através da classe ***tuple***, ou colocando valores separados por vírgula de parênteses.

Exemplo

frutas = (‘laranja’, Pera, “uva”,)

letras = tuple(“python”)

números = tuple([1, 2, 3, 4])

pais = (‘Brasil’,)

ACESSO DIRETO

frutas = (“maça”, ‘laranja’, Pera, “uva”,)

frutas[0] # maça

frutas[2] # uva

frutas[-1] #pera

frutas[-3] #laranja

TUPLAS ANINHADA

matriz = (

(1, “a”, 2),

(“b”, 3, 4),

(6, 5, “c”),

)

matriz[0] # [1, “a”, 2]

matriz[0][0] # 1

matriz[0][-1] #2

matriz[-1][-1] #c

TUPLAS FATIAMENTO

tuplas = (“P”, “y”, “t”, “h”, “o”, “n”,)

lista[2:] #(“t”, “h”, “o”, “n”)

lista[:2] #(“P”, “y”)

lista[1:3] #(“y”, “t”)

lista[0:3:2] # (“p”, “t”)

lista[::] #(“P”, “y”, “t”, “h”, “o”, “n”)

lista[::-1] #(“n”, “o”, “h”, “t”, “y”, “p”)

METODO DA CLASSE TUPLAS

().count

cores = ("vermelho", "azul", "verde", "azul",)

print(cores.count("vermelho"))    # 1

print(cores.count("azul"))  #2

print(cores.count("verde"))  #1

().index

linguagens = ("python", "js", "c", "java", "csharp",)

print(linguagens.index("java"))         # 3

print(linguagens.index("python"))   # 0

len

linguagens = ("python", "js", "c", "java", "csharp",)

print(linguagens) #>>> 6

print(len(linguagens[1])) #>>> 2

CONJUNTOS

Um set é uma coleção que possui objetos repetidos, usamos sets para representar conjuntos matemáticos ou eliminar itens duplicados de um iterável.

set([1, 2, 3, 1, 3, 4]) #{1, 2, 3, 4}

set(“abacaxi”) #{“b”, “a”, “c”, “x”, “i”}

set((“palio”, “gol”, “celta”, “palio”)) #{“gol”, “palio”, “celta”}

ACESSANDO OS DADOS

Conjuntos em Python não suportam indexação e nem fatiamento, caso queira acessar os seus valores é necessário converter o conjunto para lista.

numeros = {1, 2, 3, 2}

numeros = list(numeros)

print(numero[0])

PERCORRENDO

carros = {“gol”, “celta”, “palio”}

for carro in carros:

print(carro)

FUNÇÂO ENUMERATE

carros = {“gol”, “celta”, “palio”}

for indeice, carro in enumerate(carros):

print(f”{indice}: {carro}”)

METODO {}.UNION

Texto

Descrição gerada automaticamente

METODO {}.INTERSECTION

Texto

Descrição gerada automaticamente

METODO {}.DIFFERENCE

Texto

Descrição gerada automaticamente

METODO {}.SYMMETRIC\_DIFFERENCE

Texto

Descrição gerada automaticamente

METODO {}.ISSUBSET

Texto

Descrição gerada automaticamente

METODO {}.ISSUPERSET

Texto

Descrição gerada automaticamente

METODO {}.ISDISJOINT

Texto

Descrição gerada automaticamente

METODO {}.ADD

Texto

Descrição gerada automaticamente

METODO {}.CLEAR

Texto

Descrição gerada automaticamente

METODO {}.COPY

Texto

Descrição gerada automaticamente

METODO {}.DISCARD

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

METODO {}.POP

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

METODO {}.REMOVE

Texto

Descrição gerada automaticamente

LEN

Texto

Descrição gerada automaticamente

IN

Tela de celular com texto preto sobre fundo branco

Descrição gerada automaticamente com confiança média

DICIONARIOS

É um conjunto não-ordenado de pares chave:valor, onde as chaves são únicas em uma dada instancia do dicionário. Dicionários são delimitados por chaves: {}, e contem uma lista de pares chave:valor separada por vírgulas

Exemplo

pessoa = {“nome”: “Guilherme”, “idade”: 28}

pessoa = dict {nome= “Guilherme”, idade=28}

pessoa[“telefone”] = “3333-1234”

# >>> {“nome”: “Guilherme”, “idade”: 28 , “telefone”: “3333-1234”}

#ACESSANDO OS DADOS

dados = {“nome”: “Guilherme”, “idade”: 28 , “telefone”: “3333-1234”}

dados[“nome”] # “Guilherme”

dados[“idade”] # 28

dados[“telefone”] # “3333-1234”

dados[“nome”] = “Maria”

dados[“idade”] = 18

dados[“telefone”] = “9988-1781”

dados # {“nome”: “Maria”, “idade”: 18 , “telefone”: “9988-1781”}

#DICIONARIOS ANINHADOS

contatos = {

“gui@gmail.com”: {“nome”: “Guilherme”, “telefone”: “3333-2221”},

“carlos@gmail.com”: {“nome”: “Carlos”, “telefone”: “2929-2222”},

“leticia@gmail.com”: {“nome”: “Leticia”, “telefone”: “3350-4452”},

“bernardo@gmail.com”: {“nome”: “Bernardo”, “telefone”: “3325-1111”},

}

contatos[“carlos@gmail.com”][“telefone”] # “2929-2222”

#DICIONARIOS INTERANDO

Texto

Descrição gerada automaticamente

METODOS DA CLASSE DICT

{}.CLEAR

Texto

Descrição gerada automaticamente

{}.COPY

Texto

Descrição gerada automaticamente

{}.FROMKEYS

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

{}.GET

Texto

Descrição gerada automaticamente

{}.ITEMS

Texto

Descrição gerada automaticamente

{}.KEYS

Texto

Descrição gerada automaticamente

{}.POP

Texto

Descrição gerada automaticamente com confiança baixa

{}.POPITEM

Texto

Descrição gerada automaticamente

{}.SETDEFAULT

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

{}.UPDATE

Texto

Descrição gerada automaticamente

{}.VALUES

Texto

Descrição gerada automaticamente

In

Texto

Descrição gerada automaticamente

DEL

Texto

Descrição gerada automaticamente

FUNÇÕES

É um bloco de código identidade por um nome e pode receber uma lista de parâmetros, esses parâmetros podem ou não ter valores padrões. Usa funções torna o código mais legíveis e possibilita o reaproveitamento de código. Programar baseado em funções, é o mesmo que dizer que estamos programando de maneira estruturada.

CRIANDO A FUNÇÃO

def exibir\_mensagem():

print(“Ola mundo!”)

def exibir\_mensagem\_2(nome):

print(f“seja bem vindo {nome}!”)

def exibir\_mensagem\_3(nome=”Anonimo”):

print(f“seja bem vindo {nome}!”)

CHAMANDO A FUNÇÃO

exibir\_mensagem()

exibir\_mensagem\_2(nome=”Guilherme”)

exibir\_mensagem\_3()

exibir\_mensagem\_3(nome=”Chappie”))

RETORNANDO VALORES

Para retornar um valor, utilizamos a palavra reservada *return.*

Toda função Python retorna **None** por padrão. Diferente de outras linguagens de programação, em Python uma função pode retornar mais de um valor.

def calcular\_total(numero):

return sum(numeros)

def retorna\_antecessor\_e\_sucessor(numero)

antecessor = numero – 1

sucessor = numeo \* 1

return antecessor, sucessor

calcular\_total([10, 20, 34]): #64

retorna\_antecessor\_e\_sucessor(10) # (9,11)

ARGUMENTOS NOMEADOS

Funções também podem ser chamadas usando argumentos nomeados da forma chave=valor.

def salvar\_carro(marca, modelo, ano, placa):

#salva carro no banco de dados

print(f”Carro inserido com sucesso! {marca}/{modelo}/{ano}/{placa}”)

salvar\_carro(“Fiat”, “Palio”, 1999, “ABC-1234”)

salvar\_carro(marca=“Fiat”, modelo=“Palio”, ano=1999, placa=“ABC-1234”)

salvar\_carro(\*\*{“marca”:“Fiat”, “modelo”:“Palio”, “ano”:1999, “placa”:“ABC-1234”})

\*\* Significa que estamos passando um dicionário

ARGS E KWARGS

Podemos combinar parâmetros obrigatórios com args e kwargs. Quando esses são definidos (\*args e \*\*kwarg), o método recebe os valores como tupla e dicionário respectivamente.

Texto

Descrição gerada automaticamente

PARAMETROS ESPECIAIS

Por padrão, argumentos podem ser passados para uma função Python tanto por posição quanto explicitamente pelo nome. Para uma melhor legibilidade e desempenho, faz sentido restringir a maneira pelo qual argumentos possam ser passados, assim um desenvolvedor precisa apenas olhar para a definição da função para determinar se os itens são passados **por posição, por posição e nome, ou por nome.**

Aplicativo

Descrição gerada automaticamente com confiança baixa

POSITIONAL ONLY

Texto

Descrição gerada automaticamente

KEYWORD ONLY

Texto

Descrição gerada automaticamente

KEYWORD AND POSITIONAL ONLY

Texto

Descrição gerada automaticamente

OBJETO DE PRIMEIRA CLASSE

Em Python tudo é objeto, dessa forma **funções também são objeto** o que as tornam objetos de primeira classe. Com isso podemos **atribuir funções a variáveis, passá-las como parâmetro para funções, usá-las como valores em estruturas de dados** (listas, tuplas, dicionários etc.) e usar como valor de retorno para uma função(closures).

Texto

Descrição gerada automaticamente

ESCOPO LOCAL E ESCOPO GLOBAL

Python trabalha com escopo local e global, dentro do bloco da função o escopo é local. Portanto alterações ali feitas em objetos imutáveis serão perdidas quando o método terminar de ser executado. Para usar objetos globais utilizamos a palavra-chave **global,** que informa ao interpretador que a variável que está sendo manipulada no escopo local é global. Essa **não é uma boa prática e deve ser evitada.**

Texto, Aplicativo

Descrição gerada automaticamente