Definições

Python é uma linguagem de programação de alto nível. Há duas maneiras do sistema interpretar python: **modo interativo** e **modo script**.

O modo interativo pode ser acessado através do terminal, e é usado, geralmente, para pequenos testes.

Para acessar, abra o terminal e digite "python". Para sair, digite "exit()" ou aperte "Crtl + d".

```
python3
```

Para abrir o menu de ajuda que explica e contextualiza a linguagem e suas funções, digite o comando abaixo

```
help()
```

O modo **script** é um jeito de escrever seus programas por meio de IDEs (Integrated development environment) ou por editores de texto.

Introdução

```
print('Hello, world')
```

Visualizar valores

Para se visualizar o valor de uma variável basta escrever como no exemplo abaixo:

```
Pi = 3.1415
print(pi) ## dessa forma, você consegue ver o valor atual da varíavel :)
```

Permitir acentos em python (UTF-8)

Caso o modo script não use por padrão a codificação UTF-8, escrever o código abaixo no início do programa:

```
# -*- coding: utf-8 -*-
```

*Com isso, será permitido acentos, tais quais: à, á, ã, ä...

Comentários

Há algumas formas de se comentar em python. Destacam-se, entre elas:

```
# Para comentários em uma linha

"""Com
Três
Aspas
Triplas
Para
Comentar
Em mais
De uma linha!
"""
```

Continuar código na linha de baixo

Para continuar um código na linha de baixo, basta usar a barra invertida () ao final da linha.

```
if mes == 1 or mes == 3 or mes == 5 or mes == 7 or mes == 8 or \
        mes == 10 or mes == 12:
```

Tipo de dado

Para saber o tipo de dado de uma variável, se é número (int), string (str), booleana (bool), lista (list), tuple (tuple) e dict (dict), basta digitar o comando abaixo:

```
type("teste")

# Coloque entre aspas a varíavel que queres saber o tipo
```

Atribuições

Para se **atribuir** um valor (seja uma string, número etc.) a uma determinada variável basta usar o sinal de igual (=).

```
SUPER_XANDÃO = 'Último herói da terra'
```

Identação

A identação em Python tem um papel muito importante, pois as instruções são delimitadas pela identação. Há três níveis de identação:
Primeiro nível: código na margem esquerda.
Segundo nível: identado em 4 espaços ou 1 tabulação (tab).
Terceiro nível: identado em 8 espaços ou 2 tabulações (tab).

```
x = 0                                   # 1o. nível
while x >= 0:                           # 1o. nível
    if x == 1:                          # 2o. nível
        print("Valor de x=", x)         # 3o. nível
        x = -10                         # 3o. nível
        break                           # 3o. nível
    x += 1                              # 2o. nível
```

Função de saída

Existem várias maneiras de se expressar uma função de saída, mas, destaca-se, especialmente:

```
## Exemplo
nome = 'Super Xandão' ## atribuição de valor
ano = 1120
print(f'O {nome} é o último herói da terra, nascido em {ano}, totalizando 900
anos de existência!')
```

*Obs: em Python, para se escrever um texto, usa-se aspas simples(' '), e **NÃO** é necessário usar ponto e vírgula no final de cada código.*

Conversão de valores

Se uma variável possui um valor, ela pode ter seu tipo alterado. Ex: uma variável inteira, ser alterada para float.

Inteiro para float

```
float(3)

## ou

x = 4
float(x)
```

Float para inteiro

```
int(3.1415) ## independente do valor, será arredondado para baixo

## ou

pi = 3.1415
int(pi)
```

Arredondamentos

Para arredondarmos um tipo de dado, usamos o *round* (valor, quantidade de casas decimais a se arredondar).

```
Pi = 3.1415
round(pi, 2) """Esse '2' indica que aparecerá somente duas casas decimais a
partir da vírgula, então, ficará pi = 3.14"""
```

Strings

Há três jeitos de se comentar em python, sendo eles:

```
Irineu = 'Você não sabe, nem eu'
Irineu = "Você não sabe, nem eu"
Irineu = '''Você não sabe, nem eu'''
```

Conferir as ações que podem ser aplicadas a uma variável

Para ter acesso a uma lista de funções aplicáveis a uma variável de determinado tipo, basta digitar o *dir* seguido do *nome da variável*

```
## Exemplo
Pi = 3.1415
dir(pi)
```

Funções

Função *upper()*

É usada para deixar todas as letras de uma string em maíusculo.

```
nome = 'ifsp'
nome = nome.upper() ## Eu fiz a atribuição do valor final à variável
```

Função *lower()*

É usada para deixar todas as letras de uma string em minúsculo.

```
nome = 'IFSP'
nome = nome.lower()
```

Função *split()* (default)

Transforma todas as palavras de uma string em uma lista. Obs: os espaços em branco entre as palavras que as distinguiram entre si. Exemplo: 'suco de uva', há dois espaços, ficando uma lista composta por "suco' "de" "uva".

```
Xandão = 'Último herói da terra'
Xandão = Xandão.split()

## vai criar a seguinte lista: ['Último', 'herói', 'da', 'terra']
```

Função *split* (com símbolos diferentes)

Por padrão o que distingue as palavras entre si na função split é o espaço em branco. Contudo, se entre as palavras haver um símbolo, podemos expressar esse símbolo dentro dos parênteses do split, fazendo com que aquele símbolo indique, agora, um espaço entre as palavras.

```
Xandão = 'Último-herói-da-terra'
Xandão = Xandão.split('-')
```

Função *capitalize*

Essa função define a primeira letra da primeira palavra da lista como maiúscula, e as demais letras iniciais das demais palavras, como minúscula.

```
console = 'qual é o melhor console?'
console = console.capitalize()
```

Função *replace(x,y)*

Essa função substitui a string (x) pela nova string informada (y).

```
name = 'João, Pedro, David, Alexandre'
name = name.replace('David', 'Davi')

## desse modo, eu substituo o "David" por "Davi"
```

Função random

Essa função recebe dois valores, e sorteia um número entre a facha de números recebida. Pode ser escrita de dois jeitos.

```
## 1° modo:
from random import randint
var1 = int(input('Digite um número: '))
var2 = int(input('Digite outro número: '))
print(randint(var1, var2))

## 2° modo:
import random
var1 = int(input('Digite um número: '))
var2 = int(input('Digite outro número: '))
print(random.randint(var1, var2))
```

Função de entrada de dados

Para se escrever um dado em Python, se usa a função *input*. Contudo, esa função é **ESPECÍFICA** para strings, mas pode ser alterada se converter o valor, igual ao exemplo abaixo (age), onde é convertido de string para número.

```
name = input('Digite o nome: ')

age int(input('Digite a idade: '))
```

Exemplo combinado 1

```
name = input('Digite o nome: ')
name = name.capitalize()
print(type(name))

age = int(input('Digite a idade: '))
print(type(age))

print(f'{name} tem {age} anos')
```

Atribuição múltipla

Python permite a atribuição múltipla de variáveis (de diversos tipos), tais quais os exemplos abaixo:

```
name, age = 'Chris', 17

x, y = 1, 2

name, age = input('Digite o nome: '), int(input('Digite a idade: '))
```

Obs: também é possível visualizar várias variáveis ao mesmo tempo como o print(exemplo1, exemplo2)

Operadores aritméticos

```
num = 1 + 2 ## soma
num = 1 - 45 ## subtração
num = 5 * 8 ## multiplicação
num = 9 / 4 ## divisão
num = 10 // 7 ## divisão, que mostra somente a parte inteira, sem resto
num = 77 % 14 ## divisão, que mostra somente o resto
num = 2 ** 3 ## exponênciação
pow(2, 3) ## também é exponenciação
num = 2 ** (1/2) ## radiciação, nesse caso, para descobrir a raiz quadrada
```

Operadores relacionais (booleanos)

```
a, b = 1, 2

print(a < b) ## a menor que b
print(a > b) ## a maior que b
print(a <= b) ## b maior igual a
print(a >= b) ## a maior igual b
print(a == b) ## a e b iguais
print(a != b) ## a diferente de b
```

Operadores lógicos

Pode se usar, também, "and" e "or" para comparar mais variáveis. Exemplo:

```
a, b, c = 1, 2, 9

a < b != c

# ou

a < b and b!= c
```

Operadores de atribuição composta

```
x+=1 ## x = x + 1
y-=1 ## y = y - 1
z*=1 ## z = z * 1
a/=1 ## a = a / 1
b=1 ## b = b % 1
```

Estrutura de decisão simples

As estruturas de repetição são delimitadas por **identação**. Exibem valores booleanos (verdadeiros ou falsos).

Estrutura

```
if<condição>:
    ## bloco de instruções
print('É isso. Acabou \U+1F44D')
```

Exemplo

```
"""Exemplo: elaborar um programa que some dois números caso o usuário responda S
(sim), para somar. Se ele responder "N (não)", o programa é encerrado."""

resp = input('Deseja iniciar o programa? (S/N)')
resp = resp.upper()
if resp == 'S':
    num1 = int(input('Digite o primeiro número: '))
    num2 = int(input('Digite o segundo número: '))
    soma = num1 + num2
    print(f'O resultado da soma de {num1} + {num2} é {soma}!')
print('\nEstamos finalizando. Tenha um bom dia :) ')
```

Obs: ocorre uma delimitação a partir da linha 4 e termina na linha 7, para indicar que fazem parte do bloco if.

Estrutura para iniciar programas

```
resp = input('Deseja iniciar o programa? (S/N)')
resp = resp.upper()
if resp == 'S':

print('\nEstamos finalizando. Tenha um bom dia :) ')
```

Estrutura de decisão composta

A estrutura de decisão composta só executará algo se a condição for **verdadeira**, caso contrário, irá executar o bloco da condição *else*.

Estrutura

```
if <condição>:
    ## bloco de instrução
else:
    ## bloco de instrução
```

Exemplo

```
## Elaborar um programa que verifica um número inteiro informado é par ou ímpar.

num = int(input('Digite um número: '))

if num % 2 == 0:
    print(f'{num} é par.')
else:
    print(f'{num} é ímpar.')

print('\nAté mais!!!')
```

*Obs: não se esqueça de **Converter** o número digitado para int, pois, por padrão, a estrutura que recebe os dados (input) trabalha com strings!*

Estruturas de decisão encadeada

Essa estrutura apenas executará o bloco a qual a condição é verdadeira. Os *elif* vão dentro do *if*, e somente **UM** deles será executada, caso seja verdadeiro. É como se fosse um *else if* em linguagem C. O *else* no final será a negação do *if*, e será executado caso tanto o *if* quanto o *elif* forem falsos.

Exemplo

```
if<condição>:
    ## bloco de instruções
elif<condição>:
    ## bloco de instruções
elif<condição>:
    ## bloco de instruções
elif<condição>:
    ## bloco de instruções
elif<condição>:
    ## bloco de instruções
## ad infinitum...
else:
    ## bloco de instruções
```

Strings

Strings são sequências de caracteres

Concatenação de strings 1

As strings podem ser concatenadas com a inserção do símbolo de adição (+). Obs: para ficar com espaços entre as strings concatenadas, deixem, entre elas um espaço em branco, como '+'.

```
nome = 'Brendon'
sobrenome = 'Franco'
ultimo = 'de Oliveira'

nomeCompleto = nome + ' ' + sobrenome + ' ' + ultimo
```

Concatenação de strings 2

É possível, também, concatenar duas ou mais strings próximas sem usar o sinal de mais (+). Para isso, coloque as strings separadas por um espaço. Obs: se quiser que haja um espaço em branco entre as strings concatenadas, deixe entre aspas simples ('') entre cada palavra

```
nomeCompleto = 'Brendon'  ' '  'Franco'  ' '  'de Oliveira'
```

Repetição de strings

Em Python, podemos repetir strings e símbolos. Para isso, basta multiplicar a string ou símbolo (usando *).

```
laugh = 24 * 'K'

emote = 6 * '*' + 'Brendon' + 6 * '*'
```

Indexação de strings

Podemos indexar uma string a partir do primeiro caractere (que possui índice 0). Para tal, colocamos o nome da string seguido do número do índice entre colchetes

```
teste = 'testando123'
print(teste[5])

## será exibido a letra "n"
```

Particionamento de string

As strings podem ser particionadas, separadas em partes. O conceito é semelhante a da indexação, com a diferença de que deve-se indicar onde começa, dois pontos (:) e onde termina esse particionamento.

```
teste = 'testando123'
print(teste[1:8])
```

Substituir o valor da string

Para fazermos isso, devemos usar a função *.replace([0], x)*, onde *0* é o índice da string que contém um determinado caractere, e *x* é o caractere que irá substituir. Não obstante, esse novo valor ficará salvo como uma cópia, necessitando, assim, ser atribuído à string.

```
## Demonstração da cópia
teste = 'testando123'
teste.replace(teste[4], '4')
print('Original: ', teste)
print('Made in China: ', teste.replace(teste[4], '4'))

## Atribuição do valor
teste = 'testando123'
teste = teste.replace(teste[4], '4')
```

Saber o tamanho de uma string

Para sabermos o tamanho de uma string usamos o comando *len('string_aqui')*, se for uma string; e *len(variável_aqui)*, se for uma variável que contém uma string.

```
print(len('otorrinolaringologista'))

profissão = 'otorrinolaringologista'
print(len(profissão))
```

Listas

Em Python, podemos criar listas que podem possuir diversos tipos de variáveis dentro delas, de diversos tipos. Para declarar-las, usamos uma variável com a atribuição dos elementos da lista, entre colchetes.

Exemplo

```
lista = ['pão', 'leite', 'mortadela', 'manteiga', 'café']
```

Concatenação de listas

Assim como as strings, as listas podem ser concatenadas do mesmo modo.

```
lista1 = ['pão', 'leite']
lista2 = ['arroz', 'feijão']

listaCompleta = lista1 + lista2
```

Indexação de listas

Do mesmo modo que as strings, as listas podem ser indexadas da mesma maneira.

```
lista = ['pão', 'leite', 'manteiga']

print(lista[0])
```

Particionamento de listas

É possível, em Python, particionar uma lista do mesmo modo que particiona uma string. Define-se um ponto para começar e terminar esse particionamento.

```
lista = ['pão', 'leite', 'manteiga']

 """exibira os elementos a partir de nada até 1, nesse caso ficará uma lista com pão e leite"""
lista[: 1]

"""exibira os elementos a partir do índice 0 até o restante da lista, ficando com leite e manteiga"""
lista[1: ]
```

Substituir o valor da lista

Ao contrário das strings, nas listas **É POSSÍVEL** alterar o valor de elementos de uma lista, somente atribuindo o elemento à posição que ele ocupa.

```
lista = ['pão', 'manteiga', 'leite']
lista[1] = 'maionese'
```

Saber o tamanho de uma lista

Para sabermos o tamanho de uma lista basta usarmos *len* seguido do nome da lista entre parênteses.

```
lista = ['pão', 'manteiga', 'leite']
len(lista)
```

Acrescentar valores no final de uma lista

Para acrescentarmos valores, sejam eles números ou strings, em uma lista, basta usarmos a função *nomeDaLista.append('O_que_deseja_acrescentar')*. **Atenção:** não é necessário fazer a atribuição da variável após esse comando à variável antes do comando. Ademais, só é possível acrescentar **UM** elemento por vez com esse comando.

```
lista = ['pão', 'leite', 'arroz', 'feijão']
lista.append('café')
```

Extender uma lista

Podemos extender uma lista anexando outra lista a ela. Para tal, usamos o comando *.extend()* com a lista que será anexada a outra entre parenteses.

```
lista1 = ['pão', 'mortadela', 'maionese']
lista2 = ['arroz', 'feijão', 'ovos']

lista1.extend(lista2)
```

Inserir itens em determinadas posições em uma lista

Para fazer isso, usa-se *insert()* com o índice e elemento que se deseja adicionar a lista, entre parenteses.

```
lista = ['pão', 'mortadela', 'maionese']
lista.insert(1, 'margarina')
```

Remover itens da lista

Para remover um elemento de uma lista utilizamos o comando *.remove()*, com o nome do elemento entre parenteses.

```
lista = ['pão', 'mortadela', 'maionese']
lista.remove('maionese')
```

Remover elementos de uma lista usando o índice

É só usar o comando *.pop()* onde o índice vai entre os parenteses.

```
lista = ['pão', 'mortadela', 'maionese']
lista.pop(2)
```

Saber o índice de determinado elemento de uma lista

Usa-se o comando *.index()* com o nome do elemento entre parenteses.

```
lista = ['pão', 'mortadela', 'maionese']
print(lista.index('pão'))
```

Saber quantas vezes determinado elemento aparece em uma lista

Basta usar o comando *.count()*, com o nome do elemento entre parenteses.

```
lista = ['pão', 'mortadela', 'maionese','mortadela', 'maionese', 'pão',
'mortadela']

print(lista.count('mortadela'))
```

Organizar em ordem crescente ou decrescente as listas

Usa-se o comando *.sort()* para organizar em **ordem crescente**, e *.sort(reverse=True)* para organizar em **ordem decrescente**.

```
## crescente
lista = ['pão', 'mortadela', 'maionese','mortadela', 'maionese', 'pão',
'mortadela']
lista.sort()

## decrescente
apelidos = ['Joca', 'Lita', 'Paulinha', 'Lita', 'Jo', 'Le', 'Lita']
apelidos.sort(reverse=True)
```

Inverter os itens em uma lista

Para tal, usa-se o comando *.reverse()*.

```
alphabet = ['a', 'b', 'c', 'd', 'e']

alphabet.reverse()
```

Copiar uma lista 1

Para se copiar uma lista, basta se usar o comando *nomeLista[:]*. Essa lista será alocada na varável *nomeLista[:]*.

```
lista = ['pão', 'leite', 'mortadela', 'arroz', 'fejão']
lista[ : ]

## Pode-se, também, selecionar uma parte da lista
lista = ['pão', 'leite', 'mortadela', 'arroz', 'fejão']
lista[2:4]
## Desse modo essa lista ficará com "mortadela" e "arroz"
```

Copiar uma lista 2

Há, também, outro modo de se copiar uma lista. Usa-se o comando *.copy()*. *Obs: lembre-se de atribuir a uma variável!*

```
alphabet = ['a', 'b', 'c', 'd', 'e']

alphabet2 = alphabet.copy()
```

Limpar uma lista 1

Basta atribuirmos à lista um colchetes vazio.

```
lista = ['pão', 'ovos', 'leite']
lista = []
```

Limpar uma lista 2

Há outro possibilidade para se limpar uma lista. Para tal, usamos o comando *.clear()*.

```
lista = ['pão', 'ovos', 'leite']
lista.clear()
```

Estruturas de repetição

While

O laço *while* é repetido enquanto a condição lógica for verdadeira. A estrutura do bloco é delimitada pela identação, então fique atento!

```
while algo == qualquer_coisa:
    ## bloco de instruções
```

exibir os resultados de cada laço do while ordenadamente

Podemos, por meio do comando *print(variável, end = ',')* exibir os resultados agrupados, com uma virgula entre cada um.

```
x = 0
while x < 1000:
    x += 1
    print(x, end = ',')
```

Obs: pode haver qualquer tipo de símbolo entre os valores, basta alterar o valor contido em end.

For

Python efetua a iteração entre os elementos de uma lista, como se a variável usada para tal tornar-se cada um dos valores da lista, por vez que passa.

Estrutura básica

```
for condicao in sequencia:
    ## bloco de instruções
```

Exemplo 1

```
ListaSuperHerois = ['SUPER XANDÃO', 'Batman', 'Demolidor', 'Capitão América',
'Chapolin colorado', 'All might']

for heroi in ListaSuperHerois:
    print(heroi, len(heroi))
```

Exemplo 2

```
## encontrar se o apelido existe na lista

apelidos = ['Joca', 'Lita', 'Paulinha', 'Lita', 'Jo', 'Le', 'Lita']
cont = 0
for a in apelidos:
    if a == 'Lita':
        cont += 1
print(f'O apelido \'{a}\' foi encontrado na lista {cont} vezes.')
```

For range()

Serve para exibir uma lista que tem um começo, final e um incremento. O começo é opcional, valendo "0" se não for colocado; o incremento também é opcional, valendo "1" se não for colocado; o final é **Obrigatório**.

Exemplo

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

Range() e len()

Podemos saber o índice de uma lista, juntamente com os elementos de cada índice. Para tal, usamos o comando abaixo:

```
jogao = ['Castlevania curse of darkness', 'Detroit become human']

for i in range(len(jogao)):
    print(i, jogao[i])
```

Break, continue, pass, else

São rotinas que atuam sobre determinados blocos.

Break

Finaliza o bloco.

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, '=', x, '*', n//x)
            break
    else:
        print(n, ' é número primo!')
```

Continue

Continua no próximo laço do bloco.

```
for num in range(2, 10):
    if num % 2 == 0:
            print(f'Número par encontrado = {num}.')
            continue
    print(f'Número ímpar = {num}.')
```

Pass

Não serve para nada, mas pode ser usada para quando o programa requerer alguma função caso não haja nenhuma, a fim de evitar erros de identação.

```
print('Apenas um teste')

num = 0

if num == 0:
  pass
```

Tuplas

Tuplas são sequências ordenadas de zeros ou mais caracteres, e são **imutáveis**. Pode-se, até mesmo, colocar várias listas dentro de uma tupla. Ela é representada normalmente por *exemplo = ('algo_aqui')*

Exemplo

```
ex = (1, 2, 3, 4, 5, 6)

print(f'{ex} é do tipo {type(ex)}.')

## Há outros jeitos para se representar as tuplas. Tais quais:

ex = 1, 2, 3, 4, 5, 6

ex = 1, ## tupla com elemento único PRECISA de vírgula

ex = () ## tupla vazia
```

Conversão de tupla para lista

Para se converter uma tupla para uma lista, basta usar o comando *list(nome_da_tupla)*.

```
ex = (1, 2, 3, 4, 5, 6)

lista = list(ex)
```

Dicionários

Os dicionários são mutáveis e declarados por meio de chaves ({}). Eles são **desordenados**, logo, não possuem índices, quisá, serem fatiados. Sua forma básica é *dicio = {'A': 'amor', 'B': 'baixinho', 'C': 'coração'}*

Converter listas e tuplas para dicionários

```
tupla = (1, 2, 3)
lista = [a, b, c]

x = dict(tupla)
y = dict(lista)
```

Criando dicionários com tuplas e listas

```
t1 = ('Nina', 9989709987)
t2 = ('Pedro', 9987779987)
t3 = ('Clara', 9989706688)
print(f'Tupla t1: {t1}.')
print(f'Tupla t2: {t2}.')
print(f'Tupla t3: {t3}.')
lista = [t1, t2, t3]
print(f'\nLista: {lista}.')
telefones = dict(lista)
print(f'\nDicionário: {telefones}.')
```

Métodos de dicionários

Método *fromkeys*

Essa função permite criar um dicionário onde as chaves são elementos de listas.

```
nome = {}
print(nomes.fromkeys([4, 2]))
## será exibido: 4: none,2: none
```

```
nomes = {}
print(nomes.fromkeys([4, 2], [10, 20]))
## será exibido: {4: [10, 20], 2: [10, 20]}
```

```
nomes = {}
nomes.fromkeys(['Ana', 'Paula'], 30)
## será exibido: {'Ana': 30, 'Paula': 30}
```

Método *get*

Obtém o conteúdo associado a determinada parte de um dicionário.

```
dias = {'Janeiro': 31, 'Fevereiro': [28, 29], 'Marco': 31, 'Abril': 30}
print(f'Dicionário dias: {dias}.')
a = dias.get('Fevereiro')
b = dias.get('Marco')
c = dias.get('Abril')
d = dias.get('Junho', 'Dias não foram encontrado!!')
print(f'\nPassando a chave Fevereiro que retornou com {a}.')
print(f'\nPassando a chave Marco que retornou com {b}.')
print(f'\nPassando a chave Abril que retornou com {c}.')
print(f'\nPassando a chave Junho que retornou com {d}.')
sum(d.values())

"""
Em cada variável (a, b, c, d) é alocado um determinado elemento de um dicionário.
"""
```

Método *in*

Retorna *true* se a chave pertence ao dicionário, e *false* caso não pertença.

```
dias = {'Janeiro': 31, 'Fevereiro': [28, 29], 'Marco': 31, 'Abril': 30}
a = 'Janeiro' in dias
b = 'Junho' in dias
c = 'Abril' in dias
print(f'Janeiro existe em dias: retorna {a}.')
print(f'Junho existe em dias: retorna {b}.')
print(f'Abril existe em dias: retorna {c}.')
```

Método *.items()*

Retorna um dicionário no formato de tupla

```
alfabeto = {'A': 'Amor', 'B': 'Baixinho', 'C': 'Coração'}
print(alfabeto)
alfabeto = alfabeto.items()
print(alfabeto)
```

Método *.keys*

Retorna um dicionário no formato de listas

```
alfabeto = {'A': 'Amor', 'B': 'Baixinho', 'C': 'Coração'}
print(alfabeto)
alfabeto = alfabeto.keys()
print(alfabeto)
```

Método *.values*

Retorna uma lista com os valores das chaves do dicionário

```
alfabeto = {'A': 'Amor', 'B': 'Baixinho', 'C': 'Coração'}
print(alfabeto)
alfabeto = alfabeto.values()
print(alfabeto)
```

Método *.pop()*

Remove a chave com o valor específicado de um dicionário

```
alfabeto = {'A': 'Amor', 'B': 'Baixinho', 'C': 'Coração'}
print(alfabeto)
alfabeto.pop('A')
print(alfabeto)
```

Método *.popitem()*

Remove a ultima chave com seu valor de um dicionário

```
alfabeto = {'A': 'Amor', 'B': 'Baixinho', 'C': 'Coração'}
print(alfabeto)
alfabeto.popitem()
print(alfabeto)
```

Método *['chave'] = valor*

Permite acrescentar um novo elemento ao dicionário. Para tal, usa-se esse comando da seguinte forma: *dicio['novo_elemento'] = 2*

```
alfabeto = {'A': 'Amor', 'B': 'Baixinho', 'C': 'Coração'}
print(alfabeto)
alfabeto['D'] = 'Dramaturgo'
print(alfabeto)
```

Método *.update()*

Concatena dois dicionários

```
alfabeto = {'A': 'Amor', 'B': 'Baixinho', 'C': 'Coração'}
podium = {1: 'Mario', 2: 'Sonic', 3: 'Zelda'}
alfabeto.update(podium)
print(alfabeto)
```

Conjuntos (*set*)

Os conjuntos são uma coleção **desordenadas** de elementos, **SEM** elementos repetidos. É criado um conjunto ao se definir seus elementos entre chaves, tais quais os dicionários. Se diferem dos dicionários justamento por não possuirem chaves e valores para cada uma.

Exemplo

```
frutas = {'maça', 'pera', 'abacaxi', 'pera', 'melão', 'banana', 'melão'}
print(f'Conjunto: {frutas}.')
print(type(frutas))
```

Converter em conjuntos

```
a = [1, 2, 3]
b = set(a)
```

Subtraindo conjuntos

a = {1, 2, 3, 4, 5}
b = {2, 3, 4, 6}

c = a - b

print(c)

Unindo conjuntos

```
a = {1, 2, 3, 4, 5}
b = {2, 3, 4, 6}

c = a | b

print(c)
```

Mostrando a intersecção dos conjuntos

```
a = {1, 2, 3, 4, 5}
b = {2, 3, 4, 6}

c = a & b

print(c)
```

Funções

Funções são sequências nomeadas de instruções que pertencem uma à outra. Há, as chamadas funções *built-in*, que são funções NATIVAS do python, e que são chamadas ao digitar, por exemplo, help(), no terminal. Há, adiante, as funções criadas pelos usuários, a fim de encadear partes de um código. São essas:

Função sem retorno

```
def somar():
    x = 10
    y = 20
    print(f'x + y = {x+y}')
somar()
```

Função com retorno

```
def contar():
    c = 0
    soma = 0
    for x in range(5):
        c += 1
        soma += x
        print(f'Soma = {soma}')
    return c
contar()
```

Função com passagem de argumento

Parâmetros são variáveis declaradas na assinatura da função e tem uso exclusivo dentro do bloco da função, e quando uma função for chamada, deve-se informar o valor para cada parâmetro, os chamados argumentos.

- Parâmetros: Parâmetro é a variável que foi definida no cabeçalho da função e que será utilizada no bloco de instrução da mesma.

- Argumentos: Argumento é o valor que será passado ao invocarmos a funções.

```
def somar(a, b): # parâmetro
    x = a + b
    print(f'Soma de a = {a} + b = {b} = {x}.')
somar(10, 20) # argumento
```

```
def somar(a, b): # parâmetro
    x = a + b
    return x
print(f'Soma de a + b = {somar(10, 20)}.')
```

Função não nomeada (anônima)

É uma função declarada **sem** nome. Uma função Lambda pode receber qualquer número de argumentos, mas eles contêm apenas uma única expressão.

```
par = lambda num: num % 2
print(f'É uma função: {par}')
print(f'Passando o argumento 4 para a função par(4): retorna {par(4)}. Pois a
divisão por ele tem resto 0. Logo é par.')
```

Modularização

Módulos são arquivos de código Python cuja interface é conhecida e que podem ser importados por outros módulos. Possui suas próprias funções conhecidas. Tem como sintaxe básica o seguinte:

```
import <nome módulo>
<nome módulo>.<objeto desejado>

# ou

from <nome módulo> import <objeto desejado>
```