pyladies São Paulo

Bem-vindxs ao workshop básico de Python

Apoio:

Prêmio Mulheres TechemSampa

Google for Entrepreneurs

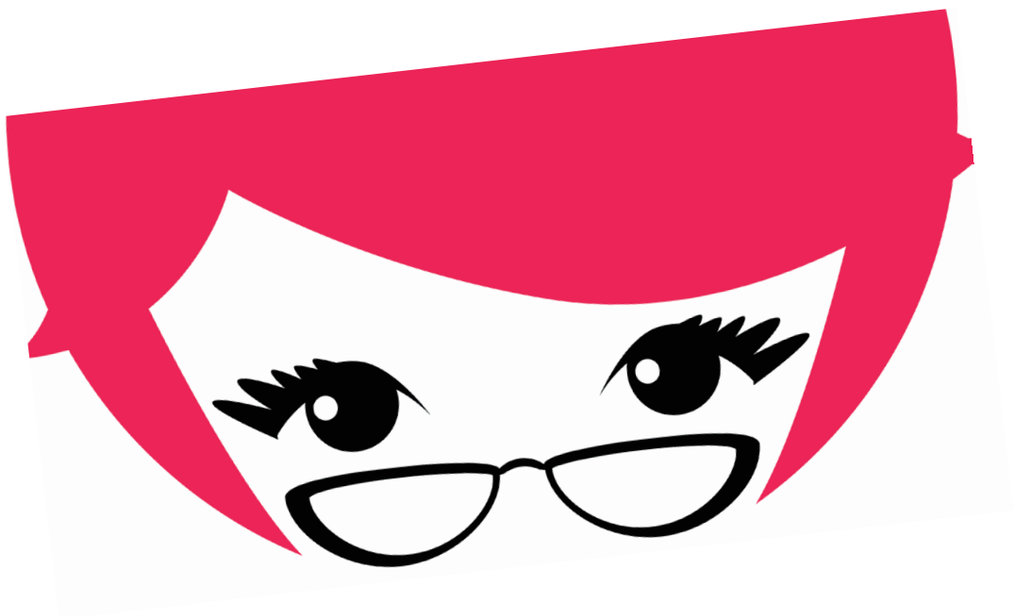
rme rede mulher empreendedora

TechSampa

O QUE É PYLADIES?

PyLadies é um grupo internacional de mentoria com foco em ajudar mais mulheres a tornarem-se participantes ativas e líderes da comunidade Python.

pyladies
São Paulo



#souPyLadiesSP

CONCEITOS BÁSICOS DE PROGRAMAÇÃO

- O que é um algoritmo?
- Para que serve?

O QUE É PYTHON?

- Linguagem de programação com código aberto de alto nível
- Multiplataforma
- Multiparadigma (imperativa, orientada a objetos, funcional)
- Dinamicamente tipada

É tipicamente usada para aplicações web, linguagens de scripts para administração de sistemas, hacking, testes de segurança de redes etc.

PRIMEIRO PROGRAMA EM PYTHON

ESCREVER NA TELA

- Para escrever nosso primeiro programa iremos utilizar a função **print( )** para imprimir um texto na tela.

```
>>> print('H3LL0, São Paulo!')
H3LL0, São Paulo
```

VARIÁVEIS EM PYTHON

- Para atribuir um valor para uma variável em Python, utilizamos a seguinte sintaxe:

```
<nome da variável> = <valor que quero armazenar>
```

- **Exemplo**:

```
>>>  a = 5
```

ALGUMAS REGRAS

Algumas regras para atribuição de variáveis:

- Python é case sensitive, isto é, diferencia maiúsculas de minúsculas
- podem ser usados algarismos, letras ou _
- nunca devem começar com um algarismo
- não podemos usar palavras-chave naturais ao Python, por exemplo if, while, etc.

Para a lista completa:

```
>>> import keyword
>>> print(keyword.kwlist)
```

ENDEREÇAMENTO

Ao utilizarmos a função id com uma variável é retornado o endereço na memória onde ela está armazenada.

- **Sintaxe**:

```
id(<variável>)
```

- **Exemplo**:

```
>>> a = 5
>>> id(a)
4297370816
>>> id(5)
4297370816
```

ENDEREÇAMENTO

Ao atribuir uma nova variável com o mesmo valor de uma variável que já está na memória, Python cria apenas uma “etiqueta” para a variável, eliminando assim o uso de um novo espaço na memória para armazenar mesmo valor.

- **Exemplo**:

```
>>> b = 5
>>> id(b)
4297370816
```

TIPOS DE VARIÁVEIS

- **Numéricos:**
  - Tipo inteiro (int): números inteiros

```
>>> a = 2
```

  - Tipo ponto flutuante (float): números racionais (em que aparece o ponto da casa decimal ou em notação científica)

```
>>> b = 1.8
>>> c = 4E-2
```

  - Tipo complexo (complex): compreende todos os números complexos. São representados por dois números de ponto flutuante (um para a parte real e um para a parte imaginária, junto com o j)

```
>>> d = 2+3j
```

TIPOS DE VARIÁVEIS

- **Literais:**

  - **Tipo string (str)**: comprende todos os caracteres dentro de aspas (simples ou duplas)

```
>>> e = '2'
>>> f = 'PyLadies São Paulo'
```

- **Lógicos:**

  - Tipo lógico (**bool**): compreende respostas do tipo True ou False

```
>>> g = True
>>> h = False
```

TIPOS DE VARIÁVEIS

Para verificarmos o tipo de uma variável utilizamos a função type( )

- **Sintaxe**:

```
type(<nome da variável>)
```

- **Exemplo**:

```
>>> type(g)
<class 'bool'>
```

ATRIBUIÇÃO MÚLTIPLA

- Com Python, também é possível fazer atribuição múltipla de variáveis.

- **Sintaxe**:

```
<variável1>,<variável2> = <valor da variável1>, <valor
da variável2>
```

- **Exemplo**:

```
>>> a, b, c = 2, 3, 4
>>> a, b, c
(2, 3, 4)
```

```
>>> a, b = b, c
>>> a, b, c
(3, 4, 4)
```

OPERADORES

- Operadores numéricos básicos:

Adição: +

Subtração: -

Divisão: /

Multiplicação: *

Potenciação: **

Resto de uma divisão: %

OPERADORES

Exemplos:

```
>>> a = 2
>>> b = 3
>>> a + b
5
>>> b - 1
2
>>> a * b
6
```

Tipos de divisão:

```
>>> 10 / 8   (divisão de números inteiros)
1.25
>>> 10 % 8   (resto da divisão entre números inteiros)
2
```

Algumas diferenças entre versões anteriores à 3.x

esse x é qualquer versão acima do 3.0

Em versões do Python anteriores à 3.x, a divisão retornava o seguinte resultado:

```
>>> 5 / 2
2
```

apenas o resultado inteiro da divisão

Caso queria apenas o resultado inteiro de uma divisão, nas versões 3.x, utilize da seguinte forma:

```
>>> 5 // 2
2
```

apenas o resultado inteiro da divisão

OPERADORES

Também é possível realizar a concatenação de caracteres/strings utilizando o operador lógico +

- **Sintaxe:**

```
'<texto/variável>' + '<texto/variável>'
```

- **Exemplos**:

```
>>> 'PyLadies ' + 'São Paulo'
'PyLadies São Paulo'
```

OPERADORES

- **Operadores lógicos ou relacionais**: servem para fazer perguntas que possam ser respondidas com True ou False (verdadeiro/falso)

Maior que: >

Menor que: <

Maior ou igual a: >=

Menor ou igual a: <=

Idêntico: ==

Diferente de: !=

Não: not

E: and (exige que todas as condições sejam satisfeitas)

Ou: or (apenas uma das condições precisa ser satisfeita)

OPERADORES

Exemplos:

```
>>> a = 2
>>> b = 3
>>> a > b
False
>>> b > a
True
>>> a == b
False
```

```
>>> a != b
True
>>> a > b and b > a
False
>>> a > b or b > a
True
>>> not (a != b) == False
True
```

OPERAÇÕES COM STRINGS

- **Quebra de linha (\n)**:

```
>>> print('PyLadies \nSão Paulo')
PyLadies
São Paulo
```

- **Tabulação (\t)**:

```
>>> print('PyLadies \tSão Paulo')
PyLadies    São Paulo
```

- **Pular linha dentro da string (\n ou """)**:

```
>>> print("""PyLadies
São
Paulo""")
PyLadies
São
Paulo
```

OPERAÇÕES COM STRINGS

- **Uppercase**: todas as letras de uma string em maiúscula (pode operar sobre uma variável do tipo texto também)
  - **Sintaxe**:

```
'<texto>'.upper()
```

  - **Exemplo**:

```
>>> 'Batata frita'.upper()
'BATATA FRITA'
```

OPERAÇÕES COM STRINGS

- **Lowercase**: todas as letras de uma string em letra minúscula

  - **Sintaxe**:
    `'<texto>'.lower()`

  - **Exemplo**:

```
>>> 'Roadsec'.lower()
'roadsec'
```

OPERAÇÕES COM STRINGS

- **Capitalize**: coloca apenas o primeiro caractere em letra maiúscula (se for um número, ele não faz nada)

  - **Sintaxe**:

```
'<texto>'.capitalize()
```

  - **Exemplo**:

```
>>> 'workshop básico de python'.capitalize()
'Workshop básico de python'
```

OPERAÇÕES COM STRINGS

- **Title**: coloca a primeira letra de cada palavra em maiúscula
  - **Sintaxe**:

    ```
    "<texto>".title()
    ```

  - **Exemplo**:

    ```
    >>> 'bolo de cenoura'.title()
    'Bolo De Cenoura''
    ```

ÍNDICE DE UMA STRING

- **Índice em uma string**: número que indica a posição de cada caractere na string

  - **Sintaxe**:

    `<variável tipo string>[número]`

O número será 0 (zero) para o primeiro caractere na string, 1 para o segundo, 2 para o terceiro, etc.

- **Exemplo**:

```
>>> a = 'PYTHON BRASIL'
>>> a[1]
'Y'
>>> a[6]
' '
>>> a[-1]
'L'
```

0	1	2	3	4	5	6	7	8	9	10	11	12
P	Y	T	H	O	N		B	R	A	S	I	L
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

FATIAS

- **Fatias de uma string**: retorna parte da string, começando pelo primeiro índice e terminando no anterior ao segundo.
  - **Sintaxe**:
    `<variável>[índice1:índice2]`
  - **Exemplo**:

```
>>> a[7:10]
'BRA'
>>> a[-1]
'L'
>>> a[7:]
'BRASIL'
```

Quando omitimos um índice, é mostrado o caractere do extremo correspondente

FATIAS

- **Incremento de fatia**: permite pegar caracteres alternados, e até mesmo inverter uma string

  - **Sintaxe**:
    `<variável>[<índice1>:<índice2>:<passo>]`

    Passo é o número de strings que queremos pular

  - **Exemplo**:

```
>>> a[7:11:2]
'BA'
>>> a[::-1]
'LISARB NOHTYP'
```

MAIS SOBRE STRINGS

- **Tamanho da string**: conta quantos caracteres tem uma string
  - **Sintaxe**:
    `len(<string>)`
  - **Exemplo**:

```
>>> a = 'filhote'
>>> len(a)
7
```

MAIS SOBRE STRINGS

- **Comando Começa com (startswith) ou Termina com (endswith)**:
  testa se um texto começa/termina com um elemento (é um teste lógico)

  - **Sintaxe**:

```
<variável a averiguar>.startswith('elemento que procuro')
```

ou

```
<variável a averiguar>.endswith('elemento que procuro')
```

MAIS SOBRE STRINGS

- **Exemplo**:

```
>>> zen_python = 'Simples é melhor que complexo.'
>>> zen_python.startswith('S')
True
>>> zen_python.startswith('s')
False
>>> zen_python.endswith('.')
True
```

Python é case sensitive, diferencia maiúscula de minúscula

MAIS SOBRE STRINGS

- **Comando Find**: procura uma string dentro de um texto e retorna a posição de seu primeiro caractere. Se houver mais que uma string igual, ele retorna a posição da primeira. Se não encontrar, ele retorna -1.

  - **Sintaxe**:

```
 <variável que contém o texto/string>.find('string
que procuro')
```

ou

```
 <variável que contém o texto/string>.find('string
que procuro', <posição a partir da qual quero
procurar>)
```

MAIS SOBRE STRINGS

- **Exemplo**:

```
>>> zen = 'Erros nunca devem passar silenciosamente. A
menos que sejam explicitamente silenciados.'
>>> zen.find('d')
12
>>> zen.find('.', 50)
86
>>> zen.find('z')
-1
```

retornou a posição da primeira ocorrência a partir do início

retornou a posição da primeira ocorrência a partir da posição 50

retorna -1 se não encontra o caracter

MAIS SOBRE STRINGS

- **Comando Replace**: troca uma string por outra dentro de um texto.

  - **Sintaxe**:

```
<variável>.replace('string que quero mudar', 'nova string')
```

  - **Exemplo**:

```
>>> x = 'Telegram é melhor que ...!'
>>> x.replace('...', 'WhatsApp')
'Telegram é melhor que WhatsApp!'
```

MARCADORES DE VARIÁVEIS

- Servem para referenciar uma variável dentro de um print.

- **Exemplos**:

Para inteiro:

```
>>>a = 3.0112
>>> print('Resultado = {}' .format(int(a)))
Resultado = 3
```

Para float:

```
>>> b = 3.0112
>>> print('Resultado = {}' .format(b))
Resultado = 3.0112
```

Para string:

```
>>> c = '3.0112'
>>> print('Resultado = {}' .format(c))
Resultado = 3.0112
```

MARCADORES DE VARIÁVEIS

Para quantidade de casas decimais:

```
>>> d = 3.0112
>>> print('Resultado = {:.2f}' .format(d))
Resultado = 3.01
```

mostra duas casas após a vírgula

Para data:

```
>>> from datetime import datetime
>>> '{:%Y-%m-%d %H:%M}'.format(datetime(2015, 9, 7, 21, 0))
'2015-09-07 21:00'
```

Para mais exemplos e possibilidades veja: https://pyformat.info/

LISTAS

- **Listas**: permitem armazenar várias informações diferentes (número, string, lógico) em uma mesma variável.

  São divididas em duas categorias, aquelas que podem ser editadas (chamadas apenas de **listas**) e as que não são editáveis (chamadas de **tuplas**).

- **Sintaxe (listas editáveis ou simplemente LISTAS):**

```
<variável> = [info1, info2, info3]
```

LISTAS

- **Exemplo 1**:

```
>>> a = ['Gato', 9, True]
>>> print(a[0])
Gato
```

Para chamar um dos elementos, uso o índice da mesma entre colchetes como faço com strings

- **Exemplo 2**:

```
>>> a = ['Gata', 5, True, ['filhotes', 3]]
>>> a[3]
['filhotes', 3]
```

O elemento 3 da lista a é uma outra lista

Exemplo de lista com um string, um inteiro, um booleano e uma lista

LISTAS

- **Comando Append**: acrescenta dados a uma lista.
  - **Sintaxe**:

    `<variável1>.append(<variável2>)`

  - **Exemplo**:

```
>>> a.append('Catnip')
>>> print(a)
['Gata', 5, True, ['filhotes', 3], 'Catnip']
```

- **Comando Join**: gruda os elementos de uma sequência de strings, usando um parâmetro fornecido

  - **Sintaxe**:

```
'<parâmetro que quero usar>'.join(<nome da sequência>)
```

- **Exemplo**:

```
>>> endereco = ['Zona Leste', 'São Paulo',
'SP']
>>> '-'.join(endereco)
'Zona Leste - São Paulo - SP'
```

Atenção: esse comando não muda a variável; se precisar usar o resultado depois, é necessário criar uma nova variável

LISTAS

- **Comando Split**: separa uma string em pontos onde existam separadores de texto (espaço, tab, enter, '/', +, etc.), criando uma lista de strings.

  - **Sintaxe**:

    ```
    '<string>'.split('<separador>')
    ```

  - **Exemplo**:

    ```
    >>> 'nome;email;documento'.split(';')
    ['nome', 'email', 'documento']
    ```

TUPLAS

- **Tuplas**: são similares às listas, mas imutáveis. Não podemos adicionar ou modificar nenhum de seus elementos.

  - **Sintaxe**:

```
<variável> = (info1, info2, info3)
```

ou

```
<variável> = info1, info2, info3
```

TUPLAS

- **Exemplo 1**:

```
>>> a = (3,5,8)
>>> a
(3,5,8)
>>> b = 3,5,8
>>> b
(3,5,8)
>>> a == b
True
>>> type(b)
<class 'tuple'>
```

Exemplo de uma tupla com um string, um inteiro, um booleano, uma lista e outra tupla

- **Exemplo 2**:

```
>>> gatos = ('Filhotes', 3, ['Zona Leste', 'São Paulo', 'SP'])
>>> gatos[2]
['Zona Leste', 'São Paulo', 'SP']
```

O elemento 2 da tupla 'gatos' é uma lista

TUPLAS

- **Sintaxe para tupla de um só elemento**:

  `<variável> = (<elemento>,)`

- **Exemplos**:

```
>>> a = (1,)
>>> a
(1,)
>>> type(a)
<class 'tuple'>
```

```
>>> a = (1)
>>> a
1
>>> type(a)
<class 'int'>
```

Os exemplos parecem iguais, mas são reconhecidos de forma diferente pelo Python

TUPLAS != LISTAS

Uma lista que pode ser alterada posteriormente ocupa mais espaço (bits) na memória, devido suas possibilidades de alteração.

Então quando usamos uma tupla, estamos economizando espaço.

Obviamente, que se formos usar uma variável que queremos aumentar ao longo do código, é melhor usar listas mutáveis.

DICAS

- **Comando dir**: lista os comandos válidos para a variável
  - **Sintaxe**:

    `dir(<variável>)`

- **Comando help**: mostra o que o comando faz com a variável
  - **Sintaxe**:

    `help(<comando aplicado à variável>)`

ENTRADA DE DADOS

- **Entrada de dados:** serve para armazenar em uma variável uma informação solicitada ao usuário.

- **Sintaxe**:

```
<variável> = input(<"mensagem para o usuário entrar com o dado>")
```

- **Exemplo:**

```
nome = input('Qual é o seu nome? ')
Qual é o seu nome? |
```

Mas o **input()** sempre retorna uma string (ele entende que o usuário digitou um texto). Se quero a entrada de um número, então preciso transformar a variável que foi lida como string em
uma variável tipo numérico (usando o **int( )**, por exemplo).

```
idade = int(input('Quantos anos você tem? '))
```

```
Quantos anos você tem? |
```

CONDICIONAIS (IF / ELSE / ELIF)

- **Sintaxe:**

```
if ____ <condição dada por operador booleano>_____:
    <o que tenho que fazer, caso a condição seja satisfeita>
else:
    <o que tenho que fazer, caso a condição não seja satisfeita>
```

- **Exemplo:**

```
semaforo = 'vermelho'
if semaforo == 'vermelho':
    print('Parar')
else:
    print('Continue pedalando')
```

CONDICIONAIS

Sintaxe:

```
if ____ <condição dada por operador booleano>_____:
    <o que tenho que fazer, caso a condição seja satisfeita>
```

(caso a condição anterior não seja satisfeita, tenho mais uma condição para verificar)

```
elif  ____<condição dada por operador booleano>_____:
    <o que tenho que fazer se a segunda condição for satisfeita>
else:
    <o que tenho que fazer, caso nenhuma das condições acima
    sejam satisfeitas>
```

CONDICIONAIS

- **Exemplo:**

```
semaforo = 'verde'
if semaforo == 'vermelho':
    print('Parar')
elif semaforo == 'amarelo':
    print('Diminua a velocidade')
else:
    print('Continue pedalando')
```

ESTRUTURAS DE REPETIÇÃO (WHILE)

- **while** é usado quando precisamos repetir uma ação algumas vezes ou fazer uma iteração até confirmar uma condição.

- **Sintaxe:**

```
while <condição a ser verificada>:
    <comando que quero executar>
```

ESTRUTURAS DE REPETIÇÃO (WHILE)

Exemplo: soma de 3 números dados pelo usuário.

```
i = 1
soma = 0
while i <= 3:
    nums = int(input('entre com um número: '))
    soma = soma + nums # soma um valor diferente a
cada vez que passa por esta linha
    i = i + 1 #contador: soma um valor fixo
print(soma)
```

INTERROMPENDO A REPETIÇÃO

Para interromper uma repetição no meio de um processo utilizamos o **Break**.

- **Exemplo:** soma de números inteiros até ser digitado zero

```
soma = 0
while True:
    x = int(input('Digite o número: '))
    if x == 0:
        break
    soma = soma +x
print('Soma: {}' .format(soma))
```

ESTRUTURAS DE REPETIÇÃO (FOR)

O comando **for** itera sobre os itens de qualquer tipo de sequência (lista ou string), na ordem em que eles aparecem na sequência. A variável que aparece na linha do **for** se comporta como cada item da lista.

- **Sintaxe**:

```
for <variável> in <lista>:
    <comando que quero executar>
```

- **Exemplo 1**:

```
linguagens = ['Java', 'JavaScript', 'PHP', 'C',
'Python']

for i in linguagens:
    if i.startswith('P'):
        print(i.upper())
```

FUNÇÕES

Funções são sub-rotinas no código que servem para executar um procedimento muitas vezes, evitando que você tenha que reescrevê-lo mais de uma vez. Elas podem apenas executar um procedimento ou retornar um valor.

- **Sintaxe**:

  Quando a função não recebe parâmetros:

```
def <nome da função> ():
```

ou

  Quando a função recebe parâmetros:

```
def <nome da função> (<parâmetro(s)>):
      <comando que quero executar>
      …
      return (caso essa função retorne algum valor)
```

FUNÇÕES

- **Exemplo**:

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

```
print(soma(1, 2))
>>> 3

#ou

print(soma('PyLadies', ' São Paulo'))
>>>
PyLadies São Paulo
```

usando como parâmetro strings

DICIONÁRIOS

Um **dicionário** é uma coleção não ordenada de pares chave-valor.

- **Sintaxe**:

```
variável = {} para dicionário vazio
variável = {'chave 1': <valor 1>, 'chave 2': <valor 2>, …,
'chave n': <valor n>}
```

- **Exemplo:**

```
>>> cat = {'cor': 'branca', 'idade': 9, 'raça':
'SRD'}
>>> cat['cor']
>>> 'branca'
```

DICIONÁRIOS

- **Para mudar um valor**: basta redefinir o valor associado à chave
  - **Sintaxe:**

```
<variável>['chave'] = 'novo valor'
```

  - **Exemplo:**

```
>>> cat['raça'] = 'russo branco'
>>> cat
{'cor': 'branca', 'raça': 'russo branco',
'idade': 9}
```

- **Para adicionar um item**: basta definir o valor associado à chave
  - **Exemplo:**

```
>>> cat['sexo'] = 'fêmea'
>>> cat
{'cor': 'branca', 'sexo': 'fêmea', 'raça':
'russo branco', 'idade': 9}
```

DICIONÁRIOS

- **Removendo itens em um dicionário usando del**
  - **Sintaxe:**

```
del <variável>['chave']
```

  - **Exemplo:**

```
>>> del cat['raça']
>>> cat
{'cor': 'branca', 'sexo': 'fêmea', 'idade': 9}
```

DICIONÁRIOS

```
>>> cat['comprimento']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'comprimento'
```

Para evitar erro ao acessar o valor de uma chave inexistente:
usar a operação **get()** do dicionário:

- **Sintaxe:**

  `<dicionario>.get(<chave>, [<valor pré-definido>])`

  sendo que `<valor pré-definido>` é opcional

- **Exemplo:**

```
>>> cat = {'idade': 9, 'cor': 'branca', 'sexo':
'fêmea'}
>>> cat.get('nome')
>>> cat.get('nome', 'Gatinho sem nome')
'Gatinho sem nome'
```

DICIONÁRIOS

Para retornar os pares chave-valor: use a operação **items()** do dicionário.

- **Sintaxe:**
  `<dicionario>.items()`

- **Exemplo:**

```
>>> cat = {'nome': 'Filoca', 'mãe': True,
'filhotes': 9, 'idade': 4, 'raça': 'indefinida',
'cor': ['preta', 'branca'], 'localização':
(15,20)}
>>> cat.items()
dict_items([('mãe', True), ('idade', 4), ('nome',
'Filoca'), ('cor', ['preta', 'branca']),
('localização', (15, 20)), ('filhotes', 9),
('raça', 'indefinida')])
```

DICIONÁRIOS

Para retornar as chaves: use a operação **keys()** do dicionário.

- **Sintaxe:**
  `<dicionario>.keys()`
- **Exemplo:**

```
>>> cat = {'nome': 'Filoca', 'mãe': True,
'filhotes': 9, 'idade': 4, 'raça': 'indefinida',
'cor': ['preta', 'branca'], 'localização':
(15,20)}
>>> cat.keys()
dict_keys(['localização', 'idade', 'nome',
'filhotes', 'raça', 'cor', 'mãe'])
```

DICIONÁRIOS

Para retornar os valores: use a operação **values()** do dicionário.

- **Sintaxe:**

```
<dicionario>.values()
```

- **Exemplo:**

```
>>> cat.values()
dict_values([(15, 20), 4, 'Filoca', 9,
'indefinida', ['preta', 'branca'], True])
```

DICIONÁRIOS

Como o resultado de **keys()** e **values()** são iteráveis,
ou seja, se comportam como listas, é possível usar com `for`:

- **Exemplos:**

```
>>> cat = {'idade': 9, 'cor': 'branca', 'sexo':
'fêmea'}
>>> for chave in cat.keys():
...     print(chave)
cor
idade
sexo
>>> for valor in cat.values():
...     print(valor)
branca
9
fêmea
```

DICIONÁRIOS

Como o resultado de **items()** também é um iterável, ou seja, se comporta como lista, é possível usar com **for**:

- **Exemplo:**

```
>>> cat = {'idade': 9, 'cor': 'branca', 'sexo': 'fêmea'}
>>> for chave, valor in cat.items():
...     print(chave)
...     print(valor)
idade
9
sexo
fêmea
cor
branca
```

DICIONÁRIOS

Para ter o dicionário ordenado pelas chaves, use a função **sorted()**.

- **Exemplo:**

```
for k in sorted(cat.keys()):
    print(k)
    print(cat.get(k))
```

Dica: a ordenação não funciona se houver tipos diferentes na chave ou valor

DICIONÁRIOS

Para controlar a ordem dos itens em um dicionário, você pode utilizar **OrderedDict** do módulo **collections**. Ele preserva exatamente a ordem original de inserção de dados em uma iteração. Por exemplo:

```
>>> from collections import OrderedDict
>>> cat = OrderedDict()
>>> cat['nome'] = 'Gatinho'
>>> cat['cor'] = 'cinza'
>>> cat['idade'] = 2
>>> cat
OrderedDict([('nome', 'Gatinho'), ('cor',
'cinza'), ('idade', 2)])
```

MÓDULOS E PACOTES

Os **módulos** são arquivos com extensão **.py** que contém definições e declarações (funções, variáveis, etc).

As definições em um módulo podem ser importadas em outros módulos, ou seja, reusar um código já pronto, sem fazer o "copia" e "cola", sem duplicar. Também podem ser chamados de bibliotecas.

Pacotes são um conjunto de módulos organizados hierarquicamente.

- **Sintaxe:**

```
import <nome_modulo>
```

MÓDULOS E PACOTES

A origem dos pacotes e módulos de um programa podem ser:

1. da biblioteca padrão do Python, ou seja, já está instalado ao instalar Python (https://docs.python.org/3/library/);

2. de terceiros, ou seja, que tem que ser instalado a parte (https://pypi.python.org/pypi, github, etc.);

3. os que você criou para seu programa.

Para visualizar todas as funções contidas em um módulo:

- **Exemplo:**

```
>>> import datetime
>>> dir(datetime)
['MAXYEAR', 'MINYEAR', '_EPOCH', '__builtins__', '__
cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'date', 'datetime', 'datetime_
CAPI', 'time', 'timedelta', 'timezone', 'tzinfo']
```

Para imprimir um número randômico:

```
>>> import random
>>> print(random.randint(1, 500))
```

Calendário:

```
>>> print('Calendário ~')
>>> import calendar
>>> print(calendar.prcal(int(input('Informe um
ano: '))))
```

MÓDULOS E PACOTES

Um módulo pode usar funções e/ou declarações definidas em outro módulo.

- calculadora_simples.py: contém 4 operações
- calculadora_financeira.py: contém as 4 operações + operações financeiras

```
# em calculadora_simples.py
def soma(a, b):
    return a + b

def multiplicacao(a, b):
    return a * b
```

```
# em calculadora_financeira.py
from calculadora_simples
import soma, divisao,
subtracao, multiplicacao
def valor_futuro(...):
…

def juros():
    multiplicacao(...)
...
```

```
# em calculadora_financeira2.py
import calculadora_simples

def valor_futuro(...):
   …

def juros():
    return calculadora_simples.
multiplicacao(...)
```

BIBLIOGRAFIA

- Curso Iniciante e Intemediário de Python – PyLadies São Paulo github.com/PyLadiesSP/Cursos

- http://pt.slideshare.net/ramiroluz/workshop-de-introduo-ao-python-tads-2015

- http://pt.slideshare.net/perone/introduo-bsica-a-linguagem-python

- http://pt.slideshare.net/ramiroluz/python-por-onde-comear-semana-technolgica-utfpr-2015

- Workshop básico em Python – PyLadies Campinas

NOSSOS CONTATOS

PyLadiesSP

PyLadiesSãoPaulo

@PyLadiesSP

Meetup PyLadiesSP

saopaulo@pyladies.com

PyLadiesSP


```
>>> print('Fim')
```