



pythondrops.com

Curso Básico de Python

(c) 2019 Cleuton Sampaio

Lição 3: Aprendendo a andar



Este trabalho e todos os seus exemplos, mesmo que não explicitamente citado, estão distribuídos de acordo com a licença Creative Commons Atribuição-CompartilhaIgual 4.0 Internacional. Este é o link para os termos desta licença:

https://creativecommons.org/licenses/by-sa/4.0/deed.pt_BR

Você tem o direito de:

- **Compartilhar** — copiar e redistribuir o material em qualquer suporte ou formato
- **Adaptar** — remixar, transformar, e criar a partir do material para qualquer fim, mesmo que comercial.

Esta licença é aceitável para Trabalhos Culturais Livres.

O licenciante não pode revogar estes direitos desde que você respeite os termos da licença.

De acordo com os termos seguintes:

- **Atribuição** — Você deve dar o crédito apropriado, prover um link para a licença e indicar se mudanças foram feitas. Você deve fazê-lo em qualquer circunstância razoável, mas de nenhuma maneira que sugira que o licenciante apoia você ou o seu uso.
- **CompartilhaIgual** — Se você remixar, transformar, ou criar a partir do material, tem de distribuir as suas contribuições sob a mesma licença que o original.
- **Sem restrições adicionais** — Você não pode aplicar termos jurídicos ou medidas de caráter tecnológico que restrinjam legalmente outros de fazerem algo que a licença permita.

Avisos:

- Você não tem de cumprir com os termos da licença relativamente a elementos do material que estejam no domínio público ou cuja utilização seja permitida por uma exceção ou limitação que seja aplicável.
- Não são dadas quaisquer garantias. A licença pode não lhe dar todas as autorizações necessárias para o uso pretendido. Por exemplo, outros direitos, tais como direitos de imagem, de privacidade ou direitos morais, podem limitar o uso do material.

Sumário

Comentários e docstring.....	4
Operadores.....	4
Aritméticos.....	4
Lógicos.....	5
Comparadores lógicos.....	6
Tipos de dados e variáveis.....	6
Números.....	7
NaN.....	9
Lógicos.....	10
None.....	11
Caracteres.....	11
Verificar o tipo.....	13
Variáveis multivaloradas.....	13
Exercício.....	23
Algoritmo.....	25
Se conseguir.....	26
Correção.....	26

Comentários e docstring

Em uma linha que contém o caractere “#”, tudo o que vier depois dele é considerado como comentário.

Não existem comentários de “bloco”, como em Java:

```
/*
```

Comentários

```
*/
```

Existe a tripla aspa dupla, que é utilizada para Docstring (documentação automática do Python), embora permita quebrar linhas, não é exatamente para comentários:

```
>>> def funcao():
...     """ Calcula alguma coisa
...         Isto é apenas uma documentação """
...     return 2 * 2
...
>>> print(funcao.__doc__)
Calcula alguma coisa
        Isto é apenas uma documentação
```

A documentação de uma função pode ser visualizada através da propriedade “__doc__”, como demonstramos.

Operadores

Aritméticos

- Soma: +
- Subtração: -
- Multiplicação: *
- Divisão: /
- Divisão inteira: //
- Exponenciação: **

- Resto de uma divisão inteira: %
- Raiz quadrada: math.sqrt()

Para calcular a raiz quadrada, você precisa importar o módulo “math” e usar a função: sqrt():

```
import math
print(math.sqrt(16))
4
```

Melhor seria calcular elevando à fração:

```
print(16**(1/2))
4
```

Lógicos

- Conjunção (E, ^): and
- Disjunção (OU, v): or
- Disjunção exclusiva (XOR, \underline{v}): ^
- Negação (Não, ~): Not ou ‘!’

Exemplos:

a = True, b = False, c = True

Lógica matemática	Python	Resultado
$a \wedge b$	a and b	False
$a \vee b$	a or b	True
$a \underline{\vee} b$	a ^ b	True
$\sim a$	not a	False
$\sim(a \wedge b)$	not (a and b)	True
$a \underline{\vee} c$	a ^ c	False

Também podem ser utilizados com valores numéricos:

a = 1, b = 0, c = 1

Lógica matemática	Python	Resultado
$a \wedge b$	<code>a and b</code>	0
$a \vee b$	<code>a or b</code>	1
$a \underline{\vee} b$	<code>a ^ b</code>	1
$\sim a$	<code>not a</code>	False(*)
$\sim(a \wedge b)$	<code>not (a and b)</code>	True(*)
$a \underline{\wedge} c$	<code>a ^ c</code>	0

(*) Devemos tomar cuidado pois algumas operações retornam valores lógicos, em vez de numéricos.

Comparadores lógicos

- Igualdade: `==` (dois sinais de igual juntos)
- Maior que: `>`
- Menor que: `<`
- Diferente: `!=`
- Maior ou igual que: `>=`
- Menor ou igual a: `<=`

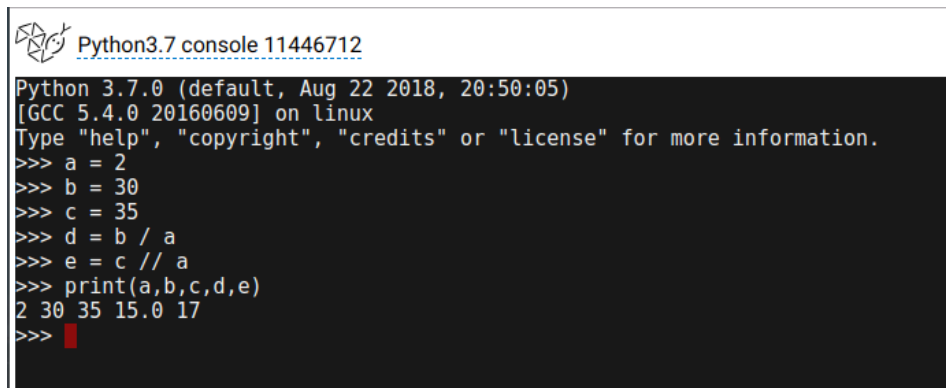
Tipos de dados e variáveis

As variáveis em Python devem ser declaradas atribuindo-se um valor a um nome. Este valor determina o tipo de dados da variável.

Abra o PythonAnywhere, Codenvy ou um terminal e vamos iniciar uma console Python, ou o Python interativo. Para isto, basta digitar: “python” (ou “python3”, se o seu computador tiver o Python 2.x instalado).

Este é o modo interativo do Python e você pode fazer quase tudo nele.

Números



```
Python3.7 console 11446712
Python 3.7.0 (default, Aug 22 2018, 20:50:05)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 2
>>> b = 30
>>> c = 35
>>> d = b / a
>>> e = c // a
>>> print(a,b,c,d,e)
2 30 35 15.0 17
>>>
```

Essas são variáveis numéricas inteiras (conjunto \mathbb{Z}), pois o valor atribuído a elas é inteiro. Note algumas coisas interessantes:

1. Ao dividir **b** por **a**, o resultado é real (há um ponto decimal), independentemente das parcelas serem inteiras;
2. O operador “//” realiza uma divisão inteira, pegando o maior número inteiro que se aproxima do resultado da divisão de **c** por **a**, retornando um valor inteiro;
3. Quando separamos os argumentos por vírgulas no **print()**, eles podem ser de tipos diferentes e podem ser números. Os argumentos são convertidos em strings e separados por espaços automaticamente;

Sei que ainda não é a hora, mas vale a pena ver algumas das características do **print()** que podem ser úteis:

```
>>> print(a + ' ' + b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

As variáveis **a** e **b** são numéricas e inteiras. Ao imprimir um string (você está concatenando as variáveis para formar um string) não é feita a conversão automática, e o Python não tem como concatenar as variáveis. Para fazer isso você deve converter os valores de **a** e **b** em string:

```
>>> print(str(a) + ' ' + str(b))
2 30
```

Por esta razão é melhor utilizar a vírgula, em vez de criar um string para imprimir.

Outra coisa interessante é: Como fazer para imprimir vários números na mesma linha? Neste caso, entra o parâmetro “end” do **print()**:

```
>>> for x in range(10):
...     print(x, end='; ')
...
0;1;2;3;4;5;6;7;8;9;>>>
```

Neste caso, cada número será separado do outro por ponto e vírgula, sem pular linha. Note que o “prompt” de comando ficou na mesma linha que o resultado.

Números reais:

```
>>> x = 7.89
>>> z = 0.1928
>>> print(x, z)
7.89 0.1928
```

As operações envolvendo números reais sempre resultarão em um número real:

```
>>> print(a, x)
2 7.89
>>> print(a + x)
9.89
>>> print(int(a + x))
9
```

A função “int()” trunca o resultado, transformando-o em um valor inteiro. A função “float()” converte o resultado em um valor real:

```
>>> a = int(2)
>>> x = float(5)
>>> print(a, x)
2 5.0
```

Não existe um limite para números inteiros, mas há um limite para o maior índice de uma lista, e pode ser obtido com a variável “sys.maxsize”:

```
>>> import sys
>>> print(sys.maxsize)
9223372036854775807
```

Este valor representa um número inteiro de 64 bits (não existe distinção entre **int** e **long** no Python 3). E podemos saber o maior número real:

```
>>> print(sys.float_info.max)
1.7976931348623157e+308
```


Python também tem um literal para infinito positivo e negativo:

```
>>> x = float('inf')
>>> x
inf
>>> z = float('-inf')
>>> z
-inf
```

Octal, hexadecimal e binário

Em Python, um literal octal começa com “0o”:

```
>>> x = 015
File "<stdin>", line 1
    x = 015
        ^
SyntaxError: invalid token
>>> x = 15
>>> x = 0o15
>>> x
13
>>>
```

Hexadecimais começam com “0x”:

```
>>> z = 0xc10
>>> z
3088
>>>
```

E, finalmente, binários começam com “0b”:

```
>>> t = 0b01101
>>> t
13
```

NaN

Assim como “inf”, Python tem um literal “NaN”, para representar um valor não numérico. O correto é que “NaN” é um valor que não pode ser representado como um número, geralmente, cálculos envolvendo infinito podem resultar em NaN. Mas divisão por zero resulta em exceção:

```
>>> x = float("inf")
>>> z = float("inf")
>>> t = x - z
>>> t
nan
>>> 5/0
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ZeroDivisionError: division by zero
```

Lógicos

Os valores True e False (assim mesmo, sem aspas e com a inicial maiúscula) definem respectivamente os valores Verdadeiro e Falso:

```
>>> b = True  
>>> v = False  
>>> print(b or v)  
True
```

Podemos utilizar os operadores lógicos de conjunção e disjunção com variáveis ou valores lógicos.

Podemos utilizar valores lógicos em condições:

```
>>> b = True  
>>> print('B verdadeiro' if b else 'B falso')  
B verdadeiro
```

Eu utilizei uma expressão lógica dentro do **print()**. É o equivalente ao “operador ternário”, existente em algumas linguagens.

Agora, veja só que interessante:

```
>>> c = 15  
>>> print('C verdadeiro' if c else 'C falso')  
C verdadeiro  
>>> d = -4  
>>> print('D verdadeiro' if d else 'D falso')  
D verdadeiro  
>>> e = 0  
>>> print('E verdadeiro' if e else 'E falso')  
E falso  
>>> f = None  
>>> print('F verdadeiro' if f else 'F falso')  
F falso
```

Qualquer valor diferente de zero e de **None** é considerado como verdadeiro!

None

Preste atenção ao valor especial None. Alguns alegam que é igual ao null, existente em outras linguagens, mas não é exatamente assim. None significa que a variável não contém valor algum, o que é diferente de variável inexistente:

```
>>> print(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g' is not defined
>>> g = None
>>> print(g)
None
```

Caracteres

Podemos criar variáveis string utilizando sequências unicode delimitadas por aspas simples ou duplas:

```
>>> nome = "João"
>>> sobrenome = 'da Silva'
>>> print(nome,sobrenome)
João da Silva
>>> print(len(nome))
4
```

A função “len()” retorna o tamanho em caracteres de um string. Porém, os strings em Python são armazenados em Unicode e podemos utilizar UTF-8 para representar texto.

Strings podem ser concatenados com o operador ‘+’:

```
>>> nome_completo = nome + ' ' + sobrenome
>>> nome_completo
'João da Silva'
```

Existem várias operações que podemos realizar com strings, por exemplo:

fatiamento (slicing):

```
>>> sobrenome
'da Silva'
>>> sobrenome[0]
'd'
>>> sobrenome[1:]
```

```
'a Silva'
>>> sobrenome[-1]
'a'
>>> sobrenome[2:-2]
'Sil'
```

O fatiamento ocorre com dois índices: inicial e final, sendo que o final geralmente é exclusivo. O índice zero representa o primeiro caractere e o índice -1 representa o último (o -2 é o penúltimo). A fatia [1:] é do segundo caractere até o final, e a fatia [2: -2] representa do terceiro caractere até o penúltimo (sem incluí-lo).

Divisão (split):

```
>>> texto = 'minha terra tem palmeiras onde canta o sabiá'
>>> palavras = texto.split()
>>> palavras
['minha', 'terra', 'tem', 'palmeiras', 'onde', 'canta', 'o', 'sabiá']
```

O método “split()” separa os tokens de um texto, delimitados por um caractere, em elementos de uma lista. Se não informarmos nada, o caractere é o espaço. Outro exemplo:

```
>>> alunos = 'João,Maria,Pedro,Augusto,Eduarda,Clara'
>>> lista = alunos.split(',')
>>> lista
['João', 'Maria', 'Pedro', 'Augusto', 'Eduarda', 'Clara']
>>> for nome in lista:
...     print(nome)
...
João
Maria
Pedro
Augusto
Eduarda
Clara
```

Procura (find):

Podemos procurar um substring dentro de um string:

```
'João da Silva'
>>> nome_completo.find('da')
5
>>> nome_completo.find('da',1,10)
5
```

O método “find()” procura um substring dentro do string, podendo verificar apenas em um intervalo (posição inicial, posição final). Ele retorna a posição do substring dentro do texto ou então o valor -1, caso não encontre.

Substituição (replace):

Podemos substituir um substring por outro (independentemente de tamanho):

```
>>> nome_completo.replace(' da ',' Da ')
'João Da Silva'
>>> nome_completo
'João da Silva'
```

Note que a substituição não ocorre “inPlace”, ou seja, o string original permanece inalterado. Podemos especificar a quantidade de vezes que a substituição deve ocorrer (o default é tudo):

```
>>> texto = 'este * texto * contém * strings'
>>> print(texto.replace('*', '', 2))
este texto contém * strings
```

Verificar o tipo

Podemos utilizar a função “type()” para saber o tipo de dados de uma variável:

```
>>> print(type(nome_completo))
<class 'str'>
>>> print(type(a))
<class 'int'>
>>> print(type(x))
<class 'float'>
>>> b = True
>>> print(type(b))
<class 'bool'>
```

Variáveis multivaloradas

Em Python temos alguns tipos de variáveis multivaloradas:

- array: conjunto de elementos homogêneos, encapsulando um vetor C;
- list: conjunto de elementos heterogêneos;
- dictionary: lista de conjuntos chave-valor indexados;
- tuple: conjunto ordenado e imutável de elementos;
- set: conjunto de elementos únicos.

Vejamos alguns exemplos:

Array:

```
>>> import array as arr
>>> x = arr.array('f', [6.5, 7.0, 6.2, 8.5, 9.7])
>>> for nota in x:
...     print('nota: {}'.format(nota))
...
nota: 6.5
nota: 7.0
nota: 6.199999809265137
nota: 8.5
nota: 9.699999809265137
>>> print('média: {}'.format(sum(x) / len(x)))
média: 7.579999923706055
```

Note o uso do método “format()” no **print()**. Há outras opções que veremos mais adiante, mas cada conjunto de chaves corresponde a um valor passado no **format()**.

Os elementos de um array devem ser do mesmo tipo, que é passado na sua declaração (primeiro argumento). Eis **alguns** tipos:

'i' / 'I'	Inteiro / Longo
'f'	Float (real)
'd'	Double (real)
'B'	Unsigned char (byte)

Usei o comando **for** para pegar cada elemento do array (diferentemente da função **range()**).

E se quiséssemos saber a ordem da nota? Poderíamos usar a opção de dois valores retornada pela função **enumerate()**:

```
>>> for i, nota in enumerate(x):
...     print('Prova: {}, nota: {}'.format(i, nota))
...
Prova: 0, nota: 6.5
Prova: 1, nota: 7.0
Prova: 2, nota: 6.199999809265137
Prova: 3, nota: 8.5
Prova: 4, nota: 9.699999809265137
```

Em Python, uma função pode retornar mais de um valor. O primeiro valor retornado pela função **enumerate()** é a variável de controle, ou índice, e a segunda é o próprio valor.

Arrays nos permitem **fatiá-los**:

```
>>> x
array('f', [6.5, 7.0, 6.199999809265137, 8.5, 9.699999809265137])
>>> print(x[1:-2])
array('f', [7.0, 6.199999809265137])
```

Neste exemplo, pegamos do segundo elemento até o penúltimo (exclusive). Funciona como o fatiamento dos strings, que já vimos.

Podemos **adicionar** ou **inserir** elementos:

```
>>> x
array('f', [6.5, 7.0, 6.199999809265137, 8.5, 9.699999809265137])
>>> x.append(6.3)
>>> x
array('f', [6.5, 7.0, 6.199999809265137, 8.5, 9.699999809265137,
6.300000190734863])
array('f', [6.5, 7.0, 6.199999809265137, 8.5, 9.699999809265137,
6.300000190734863])
>>> x.insert(1,2.5)
>>> x
array('f', [6.5, 2.5, 7.0, 6.199999809265137, 8.5, 9.699999809265137,
6.300000190734863])
```

O **append()** adiciona elementos ao final do array e o **insert()** insere elementos antes da posição informada no primeiro argumento. Temos também o **extend()** que adiciona um array ao final de outro:

```
>>> x.extend([5.5,6.1,7.2])
>>> x
array('f', [6.5, 7.0, 6.199999809265137, 8.5, 9.699999809265137,
6.300000190734863, 5.5, 6.099999904632568, 7.199999809265137])
```

E podemos **remover** elementos:

```
>>> x
array('f', [6.5, 2.5, 7.0, 6.199999809265137, 8.5, 9.699999809265137,
6.300000190734863])
>>> del x[1]
>>> x
array('f', [6.5, 7.0, 6.199999809265137, 8.5, 9.699999809265137,
6.300000190734863])
```

List:

Array quase nunca é utilizado em Python, pois a maioria dos programadores prefere a lista (list). A principal diferença é que uma lista é composta por elementos heterogêneos:

```
>>> lista = ['banana',5,True,'Volvo',5.6]
```

```
>>> for elemento in lista:
...     print('Elemento: {}'.format(elemento))
...
Elemento: banana
Elemento: 5
Elemento: True
Elemento: Volvo
Elemento: 5.6
>>> for i, elemento in enumerate(lista):
...     print('Posição: {} valor: {}'.format(i, elemento))
...
Posição: 0 valor: banana
Posição: 1 valor: 5
Posição: 2 valor: True
Posição: 3 valor: Volvo
Posição: 4 valor: 5.6
```

Basicamente existem as mesmas operações dos arrays, por exemplo, podemos anexar elementos ao final da lista:

```
>>> lista.append('Abacaxi')
>>> lista
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi']
```

Se quisermos anexar uma lista ao final, temos que tomar cuidado. Veja esse exemplo:

```
>>> lista
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi']
>>> lista.append(['Relógio', 12.7])
>>> lista
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi', ['Relógio', 12.7]]
```

O que aconteceu? Mandamos anexar uma lista ao último elemento da lista original, e ele fez isto! Uma lista é um objeto e o Python entendeu que deveríamos anexar este objeto à lista. Se quisermos estender uma lista, acrescentando vários elementos, temos que usar a função ***extend()***:

```
>>> lista
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi', ['Relógio', 12.7]]
>>> del lista[6]
>>> lista
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi']
>>> lista.extend(['Relógio', 12.7])
>>> lista
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi', 'Relógio', 12.7]
```

Removi o último elemento com o comando ***del*** e depois anexe com a função ***extend()***, e o Python entendeu corretamente.

E podemos inserir elementos e listas utilizando o fatiamento. Veja só que loucura:

```
>>> lista
```



```
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi', 'Relógio', 12.7]
>>> lista.insert(2, 'Novo')
>>> lista
['banana', 5, 'Novo', True, 'Volvo', 5.6, 'Abacaxi', 'Relógio', 12.7]
>>> lista[:2]=['uva',7]
>>> lista
['uva', 7, 'Novo', True, 'Volvo', 5.6, 'Abacaxi', 'Relógio', 12.7]
```

Eu inseri um elemento com valor “Novo” antes do elemento 2 (terceiro elemento), e depois, substituí os dois primeiros elementos (‘banana’ e 5) por: ‘uva’ e 7. Usei o fatiamento para isto. Do primeiro até o terceiro elemento exclusive.

Existem muito mais métodos em listas, por exemplo:

- `clear()`: Remove todos os elementos da lista;
- `copy()`: Retorna uma cópia da lista;
- `count()`: Retorna a quantidade de elementos na lista com o valor informado;
- `index()`: Retorna o índice na lista do elemento com o valor informado;
- `remove()`: Remove todos os elementos da lista com o valor informado;
- `reverse()`: Reverte a ordem dos elementos da lista;
- `sort()`: Classifica a lista.

Listas multidimensionais

Podemos representar vetores multidimensionais, ou matrizes, utilizando listas de listas. Por exemplo, suponha uma tabela de vendas por produto por ano por filial:

Ano	2016		2017		2018	
Filial	A	B	A	B	A	B
Centro	100	120	110	140	125	130
Sul	80	90	98	100	83	72

Podemos dizer que esta tabela tem 3 dimensões: Filial, Ano e Produto:

```
>>> vendas=[[100,120],[110,140],[125,130]],[[80,90],[98,100],[83,72]]]
>>> for filial in vendas:
...     for ano in filial:
...         for produto in ano:
...             print(produto)
...
100
120
110
140
125
130
80
90
98
100
83
72
>>> print(vendas[1][0][1])
90
```

Dictionary

Um dicionário é uma lista de pares formados por chave e valor:

```
>>> produtos = {'pizza': 50.00, 'calzone': 30.0, 'canoli':25.0}
>>> produtos
{'pizza': 50.0, 'calzone': 30.0, 'canoli': 25.0}
>>> produtos['calzone']
30.0
>>> for p in produtos:
...     print(p)
...
pizza
calzone
canoli
```

A chave de um elemento vem primeiro e depois vem o seu valor. Parece um objeto JSON (JavaScript Object Notation). Se quisermos pegar cada chave e seu correspondente valor, podemos fazer um loop assim:

```
>>> for produto,preco in produtos.items():
...     print('Produto: {}, preço: {}'.format(produto,preco))
...
Produto: pizza, preço: 50.0
Produto: calzone, preço: 30.0
Produto: canoli, preço: 25.0
```

O método items() retorna dois valores a cada passagem: A chave do item e o valor do item.

Para saber a quantidade de itens presentes em um dicionário, usamos a função **len()**. Neste caso, seria 3.

Para saber se um item existe no dicionário, podemos usar o formato “in”:

```
>>> if 'pizza' in produtos:
...     print('A pizza custa: {}'.format(produtos['pizza']))
...
A pizza custa: 50.0
```

Para adicionar e remover elementos é bem simples:

```
>>> produtos['gnocchi'] = 20.0
>>> produtos
{'pizza': 50.0, 'calzone': 30.0, 'canoli': 25.0, 'gnocchi': 20.0}
>>> produtos.pop('calzone')
30.0
>>> produtos
{'pizza': 50.0, 'canoli': 25.0, 'gnocchi': 20.0}
```

Basta atribuir um valor a uma chave inexistente e adicionamos um elemento. Para removê-lo, usamos o método “pop()” que também retorna seu valor.

Existem muito mais métodos para dicionários:

- **clear()**: Remove todos os elementos;
- **copy()**: Retorna uma cópia do dicionário;
- **fromkeys()**: Retorna um dicionário com as chaves especificadas;
- **get()**: Retorna o valor de uma chave;
- **items()**: Retorna uma lista contendo tuplas para cada elemento (chave e valor);
- **keys()**: Retorna uma lista com as chaves;
- **popitem()**: Remove o último par chave-valor inserido;
- **setdefault()**: Retorna o valor da chave especificada. Se a chave não existir, insere a chave e o valor informado no dicionário;
- **update()**: Atualiza o dicionário com os elementos informados;
- **values()**: Retorna uma lista de todos os valores no dicionário.

Tuple:

Uma tupla é um conjunto imutável de elementos. Funciona como uma lista, mas os elementos não podem ser alterados:

```
>>> cliente = ('Fulano de Tal', 'rua 5 número 7', 500)
>>> cliente
('Fulano de Tal', 'rua 5 número 7', 500)
>>> cliente[2]
500
>>> cliente[-1]
500
>>> cliente[-1] = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> if 'Fulano de Tal' in cliente:
...     print('Endereço do Fulano: {}'.format(cliente[1]))
...
Endereço do Fulano: rua 5 número 7
```

Se tentarmos alterar um valor da tupla, tomaremos um erro, como demonstrado.

Set:

Um set é um conjunto de elementos únicos, como um conjunto matemático:

```
>>> a = {'carro', 'moto', 'barco'}
>>> b = {'ônibus', 'carro', 'bicicleta'}
>>> print(a | b) # União
{'carro', 'ônibus', 'moto', 'bicicleta', 'barco'}
>>> print(a & b) # Interseção
{'carro'}
>>> print(a - b)
{'moto', 'barco'}
>>> print(b - a)
{'ônibus', 'bicicleta'}
```

Temos a união, interseção e diferença, ou seja, operações básicas de conjuntos. Note que na união, os elementos comuns não se repetem.

Um set é para operações de conjuntos, e não para acessarmos e mudarmos elementos individualmente. Portanto, não é possível acessar cada elemento como fazemos com listas. Uma maneira de acessarmos os elementos em um set é através de um loop:

```
>>> for tipo in (a | b):
...     print('O elemento é: {}'.format(tipo))
...
O elemento é: carro
```

```
0 elemento é: ônibus
0 elemento é: moto
0 elemento é: bicicleta
0 elemento é: barco
```

Outra maneira é com iterator:

```
>>> i = iter(a)
>>> for t in enumerate(i):
...     print(t)
...
(0, 'carro')
(1, 'barco')
(2, 'moto')
```

O iterador retorna uma tupla com a posição e o valor do elemento do set.

Finalmente, podemos utilizar o método “pop()”, porém ele retira o elemento do set:

```
>>> b
{'carro', 'ônibus', 'bicicleta'}
>>> b.pop()
'carro'
```

Podemos adicionar ou remover elementos do set, mas não podemos alterá-los. E nem podemos ter tipos mutáveis, como: list ou dictionary como elementos do set. Veja só este exemplo:

```
>>> a.add('avião')
>>> a
{'carro', 'barco', 'avião', 'moto'}
>>> a.discard('avião')
>>> a.discard('avião')
>>> a.remove('moto')
>>> a.remove('moto')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'moto'
```

Adicionar é simples! Basta usar o método **add()**. Para remover, temos dois métodos: **discard()** e **remove()**. A diferença é que, caso o elemento não exista no set, o remove retorna um erro e o discard não.

É claro que podemos testar a existência de um elemento com o “in”:

```
>>> b
{'ônibus', 'bicicleta'}
>>> if 'bicicleta' in b:
...     print('Sim')
...
```

Sim

E podemos saber a quantidade de elementos com o ***len()***:

```
>>> print(len(b))  
2
```

Exercício

Hora de flexionar os músculos! Bora fazer um exercício para introjectar (ôpa!) o conhecimento!

Expressões infixas e posfixas

Uma expressão aritmética infixa tem o formato:

$a+b*c^d-e$

Uma expressão aritmética posfixa, utilizada nas famosas calculadoras HP, tem o formato:

$abcd^*+e-$

Para resolver expressões aritméticas, precisamos considerar a prioridade das operações. Multiplicações e divisões são mais prioritárias que somas e subtrações. Uma das maneiras de lidar com isso é transformar a expressão infixa em posfixa, e depois executar utilizando uma pilha (estrutura LIFO – Last In First Out).

Você fará um programa que receberá uma expressão infixa, a transformará em posfixa e calculará o resultado.

Para facilitar a tarefa, você já receberá a expressão na forma de uma lista (atribua a uma variável), por exemplo:

```
expressao = [-5.0, '-', 2.0, '*', 3.0, '+', 14.0, '/', -7.0]
```

E também não haverá parêntesis, colchetes ou chaves. Tudo simples.

Esta expressão tem o resultado:

$-5 - 2 * 3 + 14 / -7 = -13$

Seu formato de posfixa é:

$-5, 2, 3, *, -, 14, -7, /, +$

Utilizando listas como pilhas

Para transformar de infixa em pósfixa e para calcular a pósfixa, você precisará utilizar pilhas. Uma pilha é uma lista LIFO: List In First Out, com as operações:

- Push: Coloca um elemento na pilha, que se torna o “topo” da pilha;
- Pop: Retorna o elemento que está no topo, diminuindo o tamanho da pilha;
- Tamanho: Retorna a quantidade de elementos na pilha;
- Top: Informa o valor do elemento que está no topo da pilha, em retirá-lo;

Em Python não temos um tipo de dados “stack” (pilha), mas podemos simulá-lo com listas:

- Push: `pilha.append(elemento);`
- Pop: `pilha.pop();`
- Tamanho: `len(pilha);`
- Top: `pilha[-1];`

Testando tipos de dados

Nossa expressão pode conter números e caracteres (operadores), por exemplo:

```
expressao = [-5.0, '-', 2.0, '*', 3.0, '+', 14.0, '/', -7.0]
```

Se o elemento for um número, você tomará determinada ação, caso contrário, tomará outra ação. Como saber o tipo do elemento? Temos a função ***type()***, que não ajuda muito, mas temos a função ***isinstance(<variável>, <tipos>)*** que pode ajudar:

```
>>> c = 5
>>> isinstance(c, (int, float))
True
>>> c = 5.5
>>> isinstance(c, (int, float))
True
>>> c = '+'
>>> isinstance(c, (int, float))
False
```


Algoritmo

Seu programa terá uma variável que contém a expressão, outra que contém sua versão posfixa e mais uma que contém a pilha. O comportamento para transformar em posfixa é este:

expressao = [-5.0, '-', 2.0, '*', 3.0, '+', 14.0, '/', -7.0]

Elemento	Ação	Pilha	Saída
-5.0	Coloca na saída	[]	-5.0
-	Se pilha vazia, então coloca na pilha	[-]	-5.0
2.0	Coloca na saída	[-]	-5.0 2.0
*	Prioridade do topo da pilha é menor, empilha	[- *]	-5.0 2.0
3.0	Coloca na saída	[- *]	-5.0 2.0 3.0
+	Prioridade do topo da pilha é maior, desempilha tudo e coloca na saída. Depois, empilha	[+]	-5.0 2.0 3.0 * -
14.0	Coloca na saída	[+]	-5.0 2.0 3.0 * - 14
/	Prioridade do tipo da pilha é menor, empilha	[+ /]	-5.0 2.0 3.0 * - 14
-7.0	Coloca na saída	[+ /]	-5.0 2.0 3.0 * - 14 - 7.0
(acabou)	Desempilha tudo e coloca na saída	[]	-5.0 2.0 3.0 * - 14 - 7.0 / +

Resultado: [-5.0, 2.0, 3.0, '*', '-', '14', '-7.0', '/', '+']

Para resolver esta expressão, também usamos pilha:

Elemento	Ação	Pilha	Resultado
-5.0	Empilha	[-5.0]	0
2.0	Empilha	[-5.0, 2.0]	0
3.0	Empilha	[-5.0, 2.0, 3.0]	0
*	Retira o operando 2 e o operando 1 da pilha e calcula, empilhando o resultado	[-5.0, 6.0]	6.0
-	Retira o operando 2 e o operando 1 da	[-11]	-11

	pilha e calcula, empilhando o resultado		
14	Empilha	[-11,14]	-11
-7.0	Empilha	[-11,14,-7.0]	-11
/	Retira o operando 2 e o operando 1 da pilha e calcula, empilhando o resultado	[-11,-2]	-2
+	Retira o operando 2 e o operando 1 da pilha e calcula, empilhando o resultado	[-13]	-13

No topo da pilha está o resultado!

Como calcular

Podemos utilizar a função ***eval(<expressão string Python>)*** para calcular, por exemplo:

```
op2 = pilha.pop() # Exemplo: -2
op1 = pilha.pop() # Exemplo: -11
x = '+'
print(eval(str(op1)+x+str(op2)))
-13
```

Ao transformar os números em strings, podemos concatenar tudo formando uma expressão aritmética Python. A função ***eval()*** calculará e retornará o resultado.

Se conseguir...

Então você tem talento! Agora, introduza os parêntesis na expressão e corrija o programa para lidar com eles!

Correção

Tente fazer! Eu sei que é difícil, mas eu já dei os algoritmos básicos e as dicas suficientes para você conseguir completar o exercício. Mesmo que não consiga terminar, tente fazer o máximo possível. Há uma correção disponível na pasta “correção”, mas sugiro que você tente. Esta correção não tem o uso de parêntesis.