ASIMOV

Números e mais em Python!

Nesta aula aprenderemos sobre os números em Python e sobre como usá-los.

Aprenderemos sobre os seguintes tópicos:

1.) Tipos de Números em Python 2.) Aritmética básica 3.) Atribuição de Objeto em Python

Tipos de números

Python tem vários "tipos" de números (literais numéricos). Nós nos concentraremos principalmente em números inteiros e números de ponto flutuante.

Inteiros são apenas números inteiros, positivos ou negativos. Por exemplo: 2 e -2 são exemplos de números inteiros.

Os números de pontos flutuantes em Python são notáveis porque eles têm um ponto decimal neles, ou usam uma exponencial (e) para definir o número. Por exemplo, 2.0 e -2.1 são exemplos de números de ponto flutuante. 4E2 (4 vezes 10 para o poder de 2) também é um exemplo de um número de ponto flutuante em Python.

Ao longo deste curso, trabalharemos principalmente com números inteiros ou tipos simples de números de flutuação.

Aqui está uma tabela dos dois tipos principais que passaremos a maior parte do tempo trabalhando com alguns exemplos:

Exemplo	Tipo
1,2, -5, 1000	Inteiros
1.2, -0.5, 2e2, 3E2	Numero Ponto-flutuante

Agora vamos começar com alguma aritmética básica.

Aritmética básica

In [1]:
```
# Soma
2+1
```

Out[1]: 3

In [2]:
```
# Subtração
2-1
```

Out[2]: 1

In [3]:
```
# Multiplicação
2*2
```

Out[3]: 4

In [4]:
```
# Divisão
3/2
```

Out[4]: 1.5

In [5]:
```
# Specifying one of the numbers as a float
3.0/2
```

Out[5]: 1.5

In [6]:
```
# Works for either number
3/2.0
```

Out[6]: 1.5

In [7]:
```
# Quadrado
2**3
```

Out[7]: 8

In [8]:
```
# Outra forma de fazer raiz quadrada
4**0.5
```

Out[8]: 2.0

In [9]:
```
# Ordem das operações em Python igual na matemática normal
2 + 10 * 10 + 3
```

Out[9]: 105

In [10]:
```
# Você pode usar parêntesis para especificar a ordem
(2+10) * (10+3)
```

Out[10]: 156

Atribuições de variáveis

Agora que vimos como usar números em Python como calculadora, vejamos como podemos atribuir nomes e criar variáveis.

Usamos um único sinal de igual para atribuir rótulos às variáveis. Vejamos alguns exemplos de como podemos fazer isso.

In [11]:
```
# Vamos criar um objeto chamado "a" e atribuir o número 5
a = 5
```

Agora, se eu chamar *a* no meu script Python, o Python tratará isso como o número 5.

In [12]:
```
# Somando objetos
a+a
```

Out[12]: 10

O que acontece na redefinição? Python nos deixa escrever sobre os valores anteriores?

In [13]:
```
# Redefinição
a = 10
```

In [14]:
```
# Checa
a
```

Out[14]: 10

In [15]:
```
# Usa a para redefinir a
a = a + a
```

In [16]:
```
# Checa
a
```

Out[16]: 20

Os nomes que você usa ao criar esses rótulos precisam seguir algumas regras:

1. Os nomes não podem começar com um número.

2. Não pode haver espaços no nome. Use _ em vez disso. 3. Não é possível usar nenhum desses símbolos: '", <> /? | \ ()! @ # $% ^ & * ~ - + 3. É considerada a melhor prática (PEP8) que os nomes são minúsculos.

Usar nomes de variáveis pode ser uma maneira muito útil de acompanhar diferentes variáveis em Python. Por exemplo:

In [17]:
```
# Use nomes de objetos para manter uma melhor relação com o que está acontece
my_income = 100

tax_rate = 0.1

my_taxes = my_income*tax_rate
```

In [18]:
```
# Mostra as taxas
my_taxes
```

Out[18]: 10.0

Então, o que aprendemos? Aprendemos alguns dos conceitos básicos dos números em

Python. Também aprendemos a fazer aritmética e usar Python como uma calculadora básica. Em seguida, o envolvemos com o aprendizado sobre a atribuição variável em Python.

Em seguida, aprenderemos sobre Strings!

ASIMOV

Strings

As strings são usadas em Python para registrar informações de texto, como nome. As cordas em Python são na verdade uma *seqüência*, o que basicamente significa que o Python acompanha cada elemento da seqüência de caracteres como uma seqüência. Por exemplo, Python entende a string "hello" como uma seqüência de letras em uma ordem específica. Isso significa que poderemos usar a indexação para pegar letras particulares (como a primeira letra ou a última letra).

Essa idéia de uma seqüência é importante em Python e nós vamos abordá-la mais tarde.

Nesta palestra, aprenderemos os seguintes tópicos:

1. Criando Strings
2. Impressão de coStringsrdas
3. Indexação e corte de strings
4. Propriedades da Cadeia de Caracteres
5. Métodos de Strings
6. Formatação de impressão

Criando uma String

Para criar uma string em Python, você precisa usar aspas simples ou aspas duplas. Por exemplo:

In [1]:
```
# Uma palavra
'hello'
```

Out[1]:
```
'hello'
```

In [2]:
```
# Uma frase inteira
'This is also a string'
```

Out[2]:
```
'This is also a string'
```

In [3]:
```
# Também é possível usar aspas duplas
"String built with double quotes"
```

Out[3]:
```
'String built with double quotes'
```

Imprimindo uma String

Usando o Jupyter Notebook com apenas uma seqüência de caracteres em uma célula emitirá automaticamente cadeias de caracteres, mas a maneira correta de exibir cadeias na sua saída é usando uma função de impressão.

In [4]:
```
# Podemos simplismente declarar uma string
'Hello World'
```

Out[4]:
```
'Hello World'
```

In [5]:
```
# Note que podemos imprimir várias strings assim
'Hello World 1'
'Hello World 2'
```

Out[5]:
```
'Hello World 2'
```

Mas a maneira correta (inclusive para outras IDEs) é utilizar o método print().

In [6]:
```
print('Hello World 1')
print('Hello World 2')
print('Use \n to print a new line')
print('\n')
print('See what I mean?')
```

```
Hello World 1
Hello World 2
Use
 to print a new line


See what I mean?
```

Nós também podemos usar uma função chamada len() para verificar o comprimento de uma string!

In [33]:
```
len('Hello World')
```

Out[33]:
```
11
```

Indexação em Strings

Sabemos que as strings são uma seqüência, o que significa que o Python pode usar índices para chamar partes da seqüência. Vamos aprender como isso funciona.

Em Python, usamos colchetes [] após um objeto para chamar seu índice. Devemos também notar que a indexação começa em 0 para Python. Vamos criar um novo objeto chamado "s" e a caminharmos através de alguns exemplos de indexação.

In [2]:
```
# Define s como uma string
s = 'Hello World'
```

In [8]:
```
# Checa
s
```

Out[8]:
```
'Hello World'
```

In [9]:
```
# Printa o objeto
print(s)
```

```
Hello World
```

Vamos começar a indexar!

In [10]:
```
# Mostra o primeiro elemento (neste caso uma letra)
s[0]
```

Out[10]:
```
'H'
```

In [22]:
```
s[1]
```

Out[22]:
```
'e'
```

In [23]:
```
s[2]
```

Out[23]:
```
'l'
```

Podemos usar um **:** para executar *corte* que pega tudo até um ponto designado. Por exemplo:

In [11]:
```
# Retorna todos elementos a partir do elemento de indice 1
s[1:]
```

Out[11]:
```
'ello World'
```

In [12]:
```
# Observe que não há mudanças no elemento s
s
```

Out[12]:
```
'Hello World'
```

In [14]:
```
# Retorna tudo até o elemento de índice 3
s[:3]
```

Out[14]:
```
'Hel'
```

Observe o corte acima. Aqui, estamos dizendo ao Python que pegue tudo de 0 a 3. Não inclui o 3º índice. Você notará muito isso em Python, onde as declarações e geralmente são no contexto "até, mas não incluindo".

In [15]:
```
# Tudo
s[:]
```

Out[15]:
```
'Hello World'
```

Também podemos usar indexação negativa para retroceder.

In [16]:
```
# Última letra (um índice antes do 0, então ele começa da parte de trás)
s[-1]
```

Out[16]: 'd'

In [18]:
```
# Pega tudo, menos a última letra
s[:-1]
```

Out[18]: 'Hello Worl'

Também podemos usar notação de índice e fatia para capturar elementos de uma seqüência com espaçamentos (o espaçamento padrão é 1). Por exemplo, podemos usar dois dois pontos em uma linha e, em seguida, um número que especifica a freqüência para capturar elementos. Por exemplo:

In [42]:
```
# Pega tudo, de 1 em 1
s[::1]
```

Out[42]: 'Hello World'

In [19]:
```
# Pega tudo, mas os espaçamentos são de 2 em 2
s[::2]
```

Out[19]: 'HloWrd'

In [21]:
```
# Pega tudo, mas com passos negativos, de trás para frente.
s[::-1]
```

Out[21]: 'dlroW olleH'

Propriedades das Strings

É importante notar que as strings têm uma propriedade importante conhecida como imutabilidade. Isso significa que, uma vez que uma string é criada, os elementos nele não podem ser alterados ou substituídos. Por exemplo:

In [48]:
```
s
```

Out[48]: 'Hello World'

In [3]:
```
# Vamos tentar mudar a primeira letra para 'x'
s[0] = 'x'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-bf1b55f1477a> in <module>()
      1 # Vamos tentar mudar a primeira letra para 'x'
----> 2 s[0] = 'x'

TypeError: 'str' object does not support item assignment
```

Observe como o erro nos diz diretamente o que não podemos fazer, alterar a atribuição do item!

Algo que podemos fazer é concatenar strings!

In [50]:
```
s
```

Out[50]:
```
'Hello World'
```

In [4]:
```
# Concatenar as strings
s + ' concatenate me!'
```

Out[4]:
```
'Hello World concatenate me!'
```

In [5]:
```
# Assim podemos redefinir completamente s
s = s + ' concatenate me!'
```

In [6]:
```
print(s)
```

```
Hello World concatenate me!
```

In [7]:
```
s
```

Out[7]:
```
'Hello World concatenate me!'
```

Podemos usar o símbolo de multiplicação para criar repetições!

In [8]:
```
letter = 'z'
```

In [9]:
```
letter*10
```

Out[9]:
```
'zzzzzzzzzz'
```

Métodos embutidos em strings

Os objetos em Python geralmente possuem métodos internos. Esses métodos são funções dentro do objeto (aprenderemos sobre isso em muito mais profundidade depois) que podem executar ações ou comandos no próprio objeto.

Chamamos métodos com um ponto e depois o nome do método. Os métodos estão na forma: objeto.método(parâmetros)

Onde os parâmetros são argumentos extras que podemos passar para o método. Não se preocupe se os detalhes não fazem 100% de sentido agora. Mais tarde, criaremos nossos próprios objetos e funções!

Aqui estão alguns exemplos de métodos internos em strings:

In [10]:
```
s
```

Out[10]:
```
'Hello World concatenate me!'
```

In [11]:
```
# Coloca toda string em caixa alta
s.upper()
```

Out[11]:
```
'HELLO WORLD CONCATENATE ME!'
```

In [12]:
```
# Caixa baixa
s.lower()
```

Out[12]:
```
'hello world concatenate me!'
```

In [13]:
```
# Divide uma string nos espaços em branco (este é o padrão)
s.split()
```

Out[13]:
```
['Hello', 'World', 'concatenate', 'me!']
```

In [14]:
```
# Divide em um elemento específico (não inclui o elemento que foi dividido)
s.split('W')
```

Out[14]:
```
['Hello ', 'orld concatenate me!']
```

Existem outros métodos do que os abrangidos aqui. Visite a seção de String avançada para descobrir mais!

Formatação de impressão

Podemos usar o método .format() para adicionar objetos formatados a instruções de impressões.

A maneira mais fácil de mostrar isso é através de um exemplo:

In [15]:
```
'Insert another string with curly brackets: {}'.format('The inserted string')
```

Out[15]:
```
'Insert another string with curly brackets: The inserted string'
```

ASIMOV

Métodos

Já vimos alguns exemplos de métodos ao aprender sobre Tipos de Estrutura de Objetos e Dados em Python. Os métodos são essencialmente funções incorporadas em objetos. Mais tarde, no curso, aprenderemos sobre como criar nossos próprios objetos e métodos usando a programação orientada a objetos (OOP) e as classes.

Os métodos irão realizar ações específicas no objeto e também podem ter argumentos, assim como uma função. Esta palestra servirá apenas como uma breve introdução aos métodos e levará você a pensar em métodos de projeto globais sobre os quais retornaremos quando chegarmos ao OOP no curso.

Os métodos estão na forma:

object.method (arg1, arg2, etc ...) Mais tarde você verá que alguns métodos têm um argumento "self" referente ao próprio objeto. Você não pode ver este argumento, mas vamos usá-lo mais tarde no curso durante as palestras OOP.

Vamos dar uma rápida olhada em um exemplo dos vários métodos que uma lista possui:

In [2]:
```
# Cria uma simples lista
l = [1,2,3,4,5]
```

Felizmente, com o iPython e o Jupyter Notebook, podemos ver rapidamente todos os métodos possíveis usando a tecla Tab. Os métodos para uma lista são:

- append
- count
- extend
- insert
- pop
- remove
- reverse
- sort

Vamos tentar alguns deles:

append() nos permite adicionar elementos ao final de uma lista:

In [3]:
```
l.append(6)
```

In [4]:
```
l
```

Out[4]:
```
[1, 2, 3, 4, 5, 6]
```

Ótimo! Agora, e o count()? O método count() irá contar o número de ocorrências de um elemento em uma lista.

In [7]:
```
# Conta quantas vezes o 2 aparece na lista
l.count(2)
```

Out[7]:
```
1
```

Você sempre pode usar Shift + Tab no Jupyter Notebook para obter mais ajuda sobre o método. Em geral, Python lhe permite usar a função help():

In [17]:
```
help(l.count)
```

```
Help on built-in function count:

count(...)
    L.count(value) -> integer -- return number of occurrences of value
```

Sinta-se livre para testar o resto dos métodos para uma lista. Mais tarde, nesta seção, seu questionário envolverá a utilização de ajuda e o Google buscando métodos de diferentes tipos de objetos!

ASIMOV

Listas

Anteriormente, ao discutir strings, introduzimos o conceito de *sequência* em Python. As listas podem ser pensadas na versão mais geral de uma *sequência* em Python. Ao contrário das strings, eles são mutáveis, o que significa que os elementos dentro de uma lista podem ser alterados!

Nesta seção, aprenderemos sobre: 1.) Criação de listas 2.) Índice e corte de listas 3.) Métodos básicos da lista 4.) Listas aninhadas 5.) Introdução ao método de Compreensão em listas

As listas são construídas com colchetes [] e vírgulas que separam cada elemento da lista.

Avançemos e vejamos como podemos construir listas!

In [1]:
```
# Atribui uma lista a uma variável chamada my_list
my_list = [1,2,3]
```

Acabamos de criar uma lista de números inteiros, mas as listas podem realmente armazenar diferentes tipos de objeto. Por exemplo:

In [2]:
```
my_list = ['A string',23,100.232,'o']
```

Assim como as strings, a função len() irá dizer-lhe quantos itens estão na sequência da lista.

In [3]:
```
len(my_list)
```

Out[3]:
```
4
```

Indexação e corte

Indexar e cortar funciona exatamente como em strings. Vamos fazer uma nova lista para nos lembrar de como isso funciona:

In [7]:
```
my_list = ['one','two','three',4,5]
```

In [10]:
```
# Pega o elemento de indice 0
my_list[0]
```

Out[10]:
```
'one'
```

In [11]:
```
# Pegue o índice 1 e tudo depois
```

```
my_list[1:]
```

Out[11]:
```
['two', 'three', 4, 5]
```

In [13]:
```
# Pega tudo até o elemento de índice 3
my_list[:3]
```

Out[13]:
```
['one', 'two', 'three']
```

Nós também podemos usar + para concatenar listas, assim como fizemos por strings.

In [14]:
```
my_list + ['new item']
```

Out[14]:
```
['one', 'two', 'three', 4, 5, 'new item']
```

Nota: Isso realmente não altera a lista original!

In [15]:
```
my_list
```

Out[15]:
```
['one', 'two', 'three', 4, 5]
```

Você teria que reatribuir a lista para tornar a mudança permanente.

In [16]:
```
# Reassign
my_list = my_list + ['add new item permanently']
```

In [18]:
```
my_list
```

Out[18]:
```
['one', 'two', 'three', 4, 5, 'add new item permanently']
```

Nós também podemos usar o * para um método de duplicação semelhante às strings:

In [20]:
```
# Make the list double
my_list * 2
```

Out[20]:
```
['one',
 'two',
 'three',
 4,
 5,
 'add new item permanently',
 'one',
 'two',
 'three',
 4,
 5,
 'add new item permanently']
```

In [4]:
```
my_list
```

Out[4]:
```
['A string', 23, 100.232, 'o']
```

Métodos de Lista Básica

Se você está familiarizado com outra linguagem de programação, você pode começar a comprar arrays em outro idioma e listas em Python. As listas em Python, no entanto, tendem a ser mais flexíveis do que arrays em outras línguas por dois bons motivos: eles não têm tamanho fixo (o que significa que não precisamos especificar o tamanho de uma lista), e eles não têm restrição de tipo fixo (como já vimos acima).

Vamos prosseguir e explore alguns métodos mais especiais para listas:

In [6]:
```
# Cria a lista
l = [1,2,3]
```

Use o método **append** para adicionar permanentemente um item ao final de uma lista:

In [7]:
```
# Append
l.append('append me!')
```

In [8]:
```
# Mostra
l
```

Out[8]:
```
[1, 2, 3, 'append me!']
```

Use **pop** para "retirar" um item da lista. Por padrão, pop tira o último índice, mas também pode especificar qual índice aparecer. Vamos ver um exemplo:

In [9]:
```
# Retira o item de índice 0
l.pop(0)
```

Out[9]:
```
1
```

In [46]:
```
# Mostra
l
```

Out[46]:
```
[2, 3, 'append me!']
```

In [10]:
```
# Atribui o elemento retirado, lembre-se de que o índice padrão é -1
popped_item = l.pop()
```

In [48]:
```
popped_item
```

Out[48]:
```
'append me!'
```

In [49]:
```
# Mostra a lista restante
l
```

Out[49]:
```
[2, 3]
```

Também deve notar-se que a indexação das listas retornará um erro se não houver nenhum elemento nesse índice. Por exemplo:

In [11]:
```
l[100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-11-e2a0c2623844> in <module>()
----> 1 l[100]

IndexError: list index out of range
```

Podemos usar o método **sort** e os métodos **reverse** para alterar suas listas:

In [13]:
```
new_list = ['a','e','x','b','c']
```

In [14]:
```
new_list
```

Out[14]:
```
['a', 'e', 'x', 'b', 'c']
```

In [15]:
```
# Use o reverso para reverter a ordem (isto é permanente!)
new_list.reverse()
```

In [16]:
```
new_list
```

Out[16]:
```
['c', 'b', 'x', 'e', 'a']
```

In [55]:
```
# Use ordenar para classificar a lista (neste caso, ordem alfabética)
new_list.sort()
```

In [56]:
```
new_list
```

Out[56]:
```
['a', 'b', 'c', 'e', 'x']
```

Listas aninhadas

Uma ótima característica das estruturas de dados do Python é que eles suportam *aninhamento* . Isso significa que podemos ter estruturas de dados dentro das estruturas de dados. Por exemplo: uma lista dentro de uma lista.

Vamos ver como isso funciona!

In [57]:
```
# Começamos com 3 listas
lst_1=[1,2,3]
lst_2=[4,5,6]
lst_3=[7,8,9]

# Faça uma lista de listas para formar uma matriz
matrix = [lst_1,lst_2,lst_3]
```

In [60]:
```
# Mostra
matrix
```

Out[60]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Agora, podemos usar novamente a indexação para pegar elementos, mas agora existem dois

níveis para o índice. Os itens no objeto matriz e, em seguida, os itens dentro dessa lista!

In [61]:
```
# Pegue o primeiro item no objeto da matriz
matrix[0]
```

Out[61]:
```
[1, 2, 3]
```

In [62]:
```
# Pegue o primeiro item do primeiro item no objeto da matriz
matrix[0][0]
```

Out[62]:
```
1
```

Compreensão de listas

Python possui um recurso avançado chamado compreensões de lista. Eles permitem a construção rápida de listas. Para entender completamente as compreensões da lista, precisamos entender os loops. Portanto, não se preocupe se você não entender completamente esta seção e sinta-se à vontade para ignorá-la, pois retornaremos a esse assunto mais tarde.

Mas no caso de você querer saber agora, aqui estão alguns exemplos!

In [63]:
```
# Crie uma lista de compreensão desconstruindo um loop for dentro de um []
first_col = [row[0] for row in matrix]
```

In [64]:
```
first_col
```

Out[64]:
```
[1, 4, 7]
```

Usamos a compreensão da lista aqui para pegar o primeiro elemento de cada linha no objeto da matriz. Vamos abordar isso com muito mais detalhes mais tarde!

Para obter métodos e recursos mais avançados das listas em Python, consulte a seção de lista avançada mais adiante neste curso!

ASIMOV

Dicionários

Nós temos aprendido sobre *sequências* em Python, mas agora vamos mudar de engrenagem e aprender sobre *mapeamentos* em Python. Se você está familiarizado com outras linguagens, pode pensar nestes Dicionários como tabelas de hash.

Esta seção servirá como uma breve introdução aos dicionários e consiste em:

1.) Construindo um Dicionário

2.) Acessando objetos de um dicionário 3.) Dicionários de assentamento 4.) Métodos básicos do dicionário

Então, o que são os mapeamentos? Os mapeamentos são uma coleção de objetos que são armazenados por uma *chave*, ao contrário de uma seqüência que armazena objetos por sua posição relativa. Esta é uma distinção importante, uma vez que os mapeamentos não reterão a ordem, pois possuem objetos definidos por uma chave.

Um dicionário de Python consiste em uma chave e depois em um valor associado. Esse valor pode ser quase qualquer objeto Python.

Construindo um Dicionário

Vamos ver como podemos construir dicionários para obter uma melhor compreensão de como eles funcionam!

In [1]:
```
# Cria um dicionário com {} e: que significa uma chave e um valor
my_dict = {'key1':'value1','key2':'value2'}
```

In [2]:
```
# Chamando valores pela chave
my_dict['key2']
```

Out[2]:
```
'value2'
```

É importante notar que os dicionários são muito flexíveis com relação aos tipos de dados que eles podem conter. Por exemplo:

In [13]:
```
my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}
```

In [4]:
```
# Vamos chamar itens do dicionário
my_dict['key3']
```

Out[4]:
```
['item0', 'item1', 'item2']
```

In [5]:
```
# Podemos chamar itens de uma lista presente na posição referente à chave 'ke
my_dict['key3'][0]
```

Out[5]:
```
'item0'
```

In [7]:
```
# Podemos chamar métodos nos itens também
my_dict['key3'][0].upper()
```

Out[7]:
```
'ITEM0'
```

Podemos também alterar valores através da chave.

In [14]:
```
my_dict['key1']
```

Out[14]:
```
123
```

In [15]:
```
my_dict['key1'] = my_dict['key1'] - 123
```

In [16]:
```
my_dict['key1']
```

Out[16]:
```
0
```

Uma nota rápida: o Python possui um método interno de fazer uma subtração ou adição automática (ou multiplicação ou divisão). Poderíamos ter usado += ou -= para a atribuição. Por exemplo:

In [17]:
```
# Define o objeto como sendo ele mesmo menos 123
my_dict['key1'] -= 123
my_dict['key1']
```

Out[17]:
```
-123
```

Também podemos criar chaves por atribuição. Por exemplo, se começássemos com um dicionário vazio, poderíamos adicionar-lhe continuamente:

In [21]:
```
# Cria um novo dicionário
d = {}
```

In [22]:
```
# Cria uma chave por associoação
d['animal'] = 'Dog'
```

In [24]:
```
# Pode fazer isso com qualquer objeto
d['answer'] = 42
```

In [25]:
```
# Mostra
d
```

Out[25]:
```
{'animal': 'Dog', 'answer': 42}
```

Aninhamento de dicionários

Espero que você esteja começando a ver o quão poderoso Python é com sua flexibilidade de objetos de nidificação e métodos dos mesmos. Vamos ver um dicionário aninhado dentro de um dicionário:

In [26]:
```
d = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

In [29]:
```
# Continue chamando as chaves...
d['key1']['nestkey']['subnestkey']
```

Out[29]:
```
'value'
```

Alguns métodos de dicionários

Existem alguns métodos que podemos chamar em um dicionário. Vamos começar uma breve introdução a alguns deles:

In [30]:
```
# Cria um dicionário típico
d = {'key1':1,'key2':2,'key3':3}
```

In [35]:
```
# Retorna uma lista de todas as chaves
d.keys()
```

Out[35]:
```
['key3', 'key2', 'key1']
```

In [36]:
```
# Pega todos os valores
d.values()
```

Out[36]:
```
[3, 2, 1]
```

In [33]:
```
# Método para retornar as tuplas de todos os itens (aprenderemos sobre as tup
d.items()
```

Out[33]:
```
[('key3', 3), ('key2', 2), ('key1', 1)]
```

Espero que você tenha agora um bom entendimento básico para a construção de dicionários. Há muito mais para explorar aqui, mas vamos revisar os dicionários mais tarde. Depois desta seção, tudo o que você precisa saber é como criar um dicionário e como recuperar seus valores.

ASIMOV

Tuplas

Em Python, as tuplas são muito semelhantes às listas, no entanto, ao contrário das listas, elas são *imutáveis*, o que significa que elas não podem ser alteradas. Você usaria tuplas para apresentar coisas que não deveriam ser alteradas, como dias da semana ou datas em um calendário.

Nesta seção, obteremos uma breve visão geral do seguinte:

1.) Construindo Tuplas 2.) Métodos básicos das Tuplas 3.) Imutabilidade 4.) Quando usar Tuplas

Você terá uma intuição de como usar tuplas com base no que você aprendeu sobre as listas. Nós podemos tratá-los de forma muito semelhante, com a maior distinção é que as tuplas são imutáveis.

Construindo Tuplas

A construção de tuplas usa () com elementos separados por vírgulas. Por exemplo:

In [1]:
```
# Pode-se criar uma tupla com múltiplos elementos
t = (1,2,3)
```

In [6]:
```
# O método len() funciona também para tuplas
len(t)
```

Out[6]: 3

In [8]:
```
# Você também pode variar os tipos de dados
t = ('one',2)

# Mostra
t
```

Out[8]: ('one', 2)

In [4]:
```
# E a indexação funciona exatamente como nas listas
t[0]
```

Out[4]: 'one'

In [11]:
```
# Corte de dados também...
t[-1]
```

Out[11]: 2

Métodos básicos da Tupla

As tuplas têm métodos internos, mas não tantas quanto listas. Vamos ver dois deles:

In [12]:
```
# Use .index com o valor de parâmetro para retornar o índice do mesmo
t.index('one')
```

Out[12]: 0

In [13]:
```
# Use .count() para saber quantas vezes determinado elemento apareceu na tupl
t.count('one')
```

Out[13]: 1

Imutabilidade

Como mencionado anteriormente, tuplas são imutáveis:

In [14]:
```
t[0]= 'change'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-93def5f9b4bd> in <module>()
----> 1 t[0]= 'change'

TypeError: 'tuple' object does not support item assignment
```

Por causa dessa imutabilidade, as tuplas não podem crescer. Uma vez que uma tupla é feita, não podemos adicionar a ela.

In [15]:
```
t.append('nope')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-799b3447c4d9> in <module>()
----> 1 t.append('nope')

AttributeError: 'tuple' object has no attribute 'append'
```

Quando usar tuplas

Você pode estar se perguntando: "Por que se preocupar em usar tuplas quando eles têm menos métodos disponíveis?" Para ser honesto, as tuplas não são usadas tantas vezes como listas na programação, mas são usadas quando a imutabilidade é necessária. Se no seu programa você está passando por um objeto e precisa ter certeza de que ele não seja alterado, então a tupla se tornará sua solução. Ele fornece uma fonte conveniente de integridade de dados.

Agora você pode criar e usar suas tuplas em sua programação, bem como ter uma compreensão da sua imutabilidade.

ASIMOV

Set e Booleanos

Existem dois outros tipos de objeto em Python que devemos cobrir rapidamente. Conjuntos(Sets) e Booleanos.

Sets

Os conjuntos são uma coleção não ordenada de elementos *únicos*. Podemos construí-los usando a função set(). Avançemos e façamos um conjunto para ver como funciona

In [3]:
```
x = set()
```

In [4]:
```
# Adicionamos elementos com o método add().
x.add(1)
```

In [5]:
```
# Mostra
x
```

Out[5]:
```
{1}
```

Observe os colchetes. Isso não indica um dicionário! Embora você possa montar analogias como um set sendo um dicionário com apenas chaves. Sabemos que um conjunto tem apenas entradas únicas. Então, o que acontece quando tentamos adicionar algo que já está em um conjunto?

In [6]:
```
# Adiciona um elemento novo
x.add(2)
```

In [7]:
```
# Mostra
x
```

Out[7]:
```
{1, 2}
```

In [8]:
```
# Adiciona o mesmo elemento
x.add(1)
```

In [10]:
```
# Mostra
x
```

Out[10]:
```
{1, 2}
```

Observe como ele não colocará mais 1 lá. Isso porque um conjunto apenas se ocupa de elementos exclusivos! Podemos transformar uma lista com múltiplos elementos repetidos para um conjunto para obter os elementos exclusivos. Por exemplo:

In [11]:
```
# Cria uma lista com elementos repetidos
l = [1,1,2,2,3,4,5,6,1,1]
```

In [12]:
```
# Transforma em um set com elementos únicos
set(l)
```

Out[12]:
```
{1, 2, 3, 4, 5, 6}
```

Booleanos

O Python possui também Booleanos (com True e False predefinidas que são basicamente apenas os números inteiros 1 e 0). Ele também possui um objeto reservado chamado None. Passemos por alguns exemplos rápidos de booleanos (vamos mergulhar mais profundamente neles mais tarde neste curso).

In [13]:
```
# Define um objeto como True
a = True
```

In [14]:
```
# Mostra
a
```

Out[14]:
```
True
```

Também podemos usar operadores de comparação para criar booleanos. Examinaremos todos os operadores de comparação mais tarde no curso.

In [15]:
```
# O output é boleano
1 > 2
```

Out[15]:
```
False
```

Nós podemos usar None como um espaço reservado para um objeto que não queremos reatribuir ainda:

In [16]:
```
b = None
```

É isso aí! Agora você deve ter uma compreensão básica de objetos Python e tipos de estrutura de dados. Em seguida, vá em frente e faça o teste de avaliação!

ASIMOV

Operadores de comparação

Nesta aula estaremos aprendendo sobre Operadores de Comparação em Python. Esses operadores nos permitirão comparar variáveis e produzir um valor booleano (Verdadeiro ou Falso).

Se você tiver alguma base em Matemática, esses operadores devem ser muito diretos.

Em primeiro lugar, apresentaremos uma tabela dos operadores de comparação e depois trabalharemos com alguns exemplos:

Tabela de operadores de comparação

Operador	Descrição	Exemplo
==	Se os valores de dois operandos forem iguais, a condição torna-se verdadeira.	(a == b) não é verdade.
!=	Se valores de dois operandos não forem iguais, a condição torna-se verdadeira.	(a != b) é verdadeiro
>	Se o valor do operando esquerdo for maior que o valor do operando direito, a condição torna-se verdadeira.	(a > b) não é verdadeiro.
<	Se o valor do operando esquerdo for inferior ao valor do operando direito, a condição torna-se verdadeira.	(a < b) é verdadeiro.
>=	Se o valor do operando esquerdo for maior ou igual ao valor do operando direito, a condição torna-se verdadeira.	(a >= b) não é verdadeiro.
<=	Se o valor do operando esquerdo for menor ou igual ao valor do operando direito, a condição torna-se verdadeira.	(a <= b) é verdadeiro

Vamos agora trabalhar com exemplos rápidos de cada um desses.

Igualdade

In [3]:
```
2 == 2
```

Out[3]:
```
True
```

In [4]:
```
1 == 0
```

Out[4]: False

Desigualdade

In [5]:
```
2 != 1
```

Out[5]: True

In [6]:
```
2 != 2
```

Out[6]: False

In [7]:
```
2 <> 1
```

Out[7]: True

In [8]:
```
2 <> 2
```

Out[8]: False

Maior que

In [9]:
```
2 > 1
```

Out[9]: True

In [10]:
```
2 > 4
```

Out[10]: False

Menor que

In [11]:
```
2 < 4
```

Out[11]: True

In [12]:
```
2 < 1
```

Out[12]: False

Maior ou igual que

In [13]:
```
2 >= 2
```

Out[13]: True

In [14]:
```
2 >= 1
```

Out[14]: True

Menor ou igual que

In [15]:
```
2 <= 2
```

Out[15]: True

In [16]:
```
2 <= 4
```

Out[16]: True

ASIMOV

Operadores de comparação em cadeia

Uma característica interessante do Python é a capacidade de *encadear* comparações múltiplas para realizar um teste mais complexo. Você pode usar essas comparações em cadeia como uma abreviatura para expressões booleanas maiores.

Nesta palestra, aprenderemos como encadear operadores de comparação e também apresentamos duas outras declarações importantes em python: **and** e **or** .

Vejamos alguns exemplos de uso de cadeias:

In [1]:
```
1 < 2 < 3
```

Out[1]: True

A declaração acima verifica se 1 era inferior a 2 **e** se 2 era inferior a 3. Poderíamos ter escrito isso usando uma instrução **and** em Python:

In [2]:
```
1<2 and 2<3
```

Out[2]: True

O **and** é usado para garantir que as duas verificações tenham que ser verdadeiras para que a verificação total seja verdadeira. Vamos ver outro exemplo:

In [3]:
```
1 < 3 > 2
```

Out[3]: True

As verificações acima checam se 3 é maior do que os outros números. Você pode usar **and** para reescrevê-lo como:

In [4]:
```
1<3 and 3>2
```

Out[4]: True

É importante notar que o Python está verificando ambas as instâncias das comparações. Nós também podemos usar **or** para escrever comparações em Python. Por exemplo:

In [5]:
```
1==2 or 2<3
```

Out[5]: True

Observe como a expressão retornou True porque com o operador **ou** precisamos apenas um

ou os outros dois sejam verdadeiros. Outro exemplo:

In [6]:
```
1==1 or 100==1
```

Out[6]:
```
True
```

Ótimo! Vá em frente e vá ao questionário para esta seção para verificar sua compreensão!

ASIMOV

Introdução às declarações do Python

Nesta palestra, faremos uma rápida visão geral das declarações do Python. Esta palestra enfatizará as diferenças entre Python e outros idiomas, como o C++.

Existem duas razões pelas quais tomamos essa abordagem para aprender o contexto das declarações de Python:

1.) Se você vem de uma linguagem diferente, isso acelerará rapidamente sua compreensão sobre o Python. 2.) Aprender sobre declarações permitirá que você possa ler outras linguagens com mais facilidade no futuro.

Python vs outras linguagens

Vamos criar uma declaração simples que diga: "Se a é maior que b, atribua 2 a **a** e 4 a **b**"

Dê uma olhada nessas duas afirmações if (aprenderemos sobre a construção de declarações if).

Versão 1 (Outras linguagens)

se (a> b) { a = 2; b = 4; } **Versão 2 (Python)**

se a> b: a = 2 b = 4

Você notará que o Python está menos confuso e muito mais legível do que a primeira versão. Como o Python gerencia isso?

Vamos percorrer as principais diferenças:

Python se livra de () e {} incorporando dois fatores principais: os *dois pontos* e os *espaços em branco*. A declaração é terminada com dois pontos e o espaço em branco é usado (recuo) para descrever o que ocorre no caso da declaração.

Outra grande diferença é a falta de ponto e vírgula em Python. Eles são usados para denotar terminações de declaração em muitos outros idiomas, mas em Python, o final de uma linha é o mesmo que o final de uma declaração.

Por fim, para terminar esta breve visão geral das diferenças, vamos examinar mais de perto a sintaxe de indentação em Python vs outros idiomas:

Indentação

Aqui está um pseudo-código para indicar o uso de espaço em branco e indentação em Python:

Outras linguagens

se (x) se (y) declaração de código; outro outra indicação de código; **Python** se x: se y: declaração de código outro: outra declaração de código

Observe como o Python é tão fortemente impulsionado por indentação de código e espaço em branco. Isso significa que a legibilidade do código é uma parte essencial do design da linguagem Python.

Agora vamos começar a mergulhar mais fundo codificando esse tipo de afirmações em Python!

ASIMOV

---

if,elif e else

If em Python nos permite contar ao computador para executar ações alternativas com base em um determinado conjunto de resultados. Verbalmente, podemos imaginar que estamos informando o computador:

"Ei, se isso caso acontecer, execute alguma ação" Podemos então expandir a idéia com declarações elif e else, o que nos permite contar ao computador:

"Ei, se esse caso acontecer, execute alguma ação. Caso contrário, se aquilo dali acontecer, execute alguma outra ação. Caso contrário, nenhum dos casos acima aconteceu, execute esta ação" Avançemos e vejamos o formato de sintaxe para as instruções if para ter uma idéia melhor disso:

if case1: executar ação1 elif case2: execute ação2 else: execute a ação 3

Primeiro exemplo

Vamos ver um exemplo rápido disso:

In [2]:
```
if True:
    print('It was true!')
```

```
It was true!
```

Vamos adicionar outra lógica:

In [3]:
```
x = False

if x:
    print('x was True!')
else:
    print('I will be printed in any case where x is not true')
```

```
I will be printed in any case where x is not true
```

Múltiplos ramos

Vamos ver de forma mais completa de quão longe if, elif, e else podem nos levar! Nós escrevemos isso em uma estrutura aninhada. Tome nota de como o if, elif e else se alinham no código. Isso pode ajudá-lo a ver o que se relaciona com o elif ou outras afirmações.

Vamos reintroduzir uma sintaxe de comparação para o Python.

In [4]:
```
loc = 'Bank'
```

```
if loc == 'Auto Shop':
    print('Welcome to the Auto Shop!')
elif loc == 'Bank':
    print('Welcome to the bank!')
else:
    print("Where are you?")
```

```
Welcome to the bank!
```

Observe como as declarações if aninhadas são verificadas até que um booleano True faça com que o código aninhado abaixo seja executado. Você também deve notar que você pode colocar as declarações elif quanto desejar antes de fechar com outra.

Vamos criar dois exemplos mais simples para as afirmações if, elif e else:

In [5]:
```
person = 'Sammy'

if person == 'Sammy':
    print('Welcome Sammy!')
else:
    print("Welcome, what's your name?")
```

```
Welcome Sammy!
```

In [6]:
```
person = 'George'

if person == 'Sammy':
    print('Welcome Sammy!')
elif person =='George':
    print("Welcome George!")
else:
    print("Welcome, what's your name?")
```

```
Welcome George!
```

Indentação

É importante manter uma boa compreensão de como o recuo funciona no Python para manter a estrutura e a ordem do seu código. Vamos voltar a tocar neste tópico quando começarmos a criar funções!

ASIMOV

---

range()

Nesta palestra curta estaremos discutindo a função range(). Ainda não desenvolvemos um nível muito profundo de conhecimento de funções, mas podemos entender o básico desta função simples (mas extremamente útil!).

range() nos permite criar uma lista de números que variam de um ponto de partida *até* um ponto final. Também podemos especificar o tamanho do passo. Vamos percorrer alguns exemplos:

In [1]:
```
range(0,10)
```

Out[1]:
```
range(0, 10)
```

In [2]:
```
x =range(0,10)
type(x)
```

Out[2]:
```
range
```

In [3]:
```
start = 0 # Por padrão
stop = 20
x = range(start,stop)
```

In [5]:
```
list(x)
```

Out[5]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Ótimo! Observe como foi *até* 20, mas na verdade não produz 20. Assim como na indexação. E o tamanho do passo? Podemos especificar isso como um terceiro argumento:

In [3]:
```
x = range(start,stop,2)
# Mostrar
x
```

Out[3]:
```
range(0, 20, 2)
```

Impressionante! Bem, é isso ... ou não?

Você pode estar se perguntando, o que acontece se eu quiser usar uma grande variedade de números? O meu computador pode armazenar tudo na memória?

Grande pensamento! Este é um dilema que pode ser resolvido com o uso de um gerador. Para uma explicação simplificada: um gerador permite a geração de objetos gerados que são fornecidos naquela instância, mas não armazena cada instância gerada na memória.

Isso significa que um gerador não criaria uma lista ao gerar um range(), mas, em vez disso, fornece uma geração única dos números nesse intervalo. A boa notítica é que range() se comporta como um gerador e você não precisa se preocupar com isso.

ASIMOV

For

Um loop **for** atua como um iterador em Python, ele passa por itens que estão em uma *sequência* ou qualquer outro item iterável. Os objetos que aprendemos até agora que podemos iterar incluem strings, listas, tuplas e até iteráveis embutidos em dicionários, como chaves ou valores.

Já vimos **for** um pouco nas palestras passadas, mas agora permitimos formalizar a nossa compreensão.

Aqui está o formato geral para um **for** loop em Python:

    for item in objeto: fazer algo

O nome da variável usado para o item fica a seu critério, você pode escolher o que quiser. Então use seu melhor julgamento para escolher um nome que faça sentido e que você poderá entender ao revisar seu código. Este nome do item pode então ser referenciado dentro de seu loop, por exemplo, se você quisesse usar instruções if para executar verificações.

Vamos seguir em frente e trabalhar com vários exemplos de **for** loops usando uma variedade de tipos de objetos de dados. Vamos começar com um exemplo simples e adicionar mais complexidade mais além.

Exemplo 1

Iterando através de uma lista.

In [1]:
```
# Aprenderemos a automatizar esse tipo de lista na próxima palestra
l = [1,2,3,4,5,6,7,8,9,10]
```

In [2]:
```
for num in l:
    print(num)
```

```
1
2
3
4
5
6
7
8
9
10
```

Ótimo! Espero que isso tenha feito sentido. Agora, vamos adicionar uma instrução if para

verificar se há números pares. Vamos primeiro apresentar um novo conceito aqui - o módulo.

Modulo

O módulo nos permite obter o restante em uma divisão e usa o símbolo %. Por exemplo:

In [5]:
```
17 % 5
```

Out[5]: 2

Isso faz sentido, pois 17 dividido por 5 é 3 e sobra 2. Vamos ver alguns exemplos mais rápidos:

In [6]:
```
# 3, sobra 1
10 % 3
```

Out[6]: 1

In [9]:
```
# 2, sobra 4
18 % 7
```

Out[9]: 4

In [10]:
```
# 2, sem sobras
4 % 2
```

Out[10]: 0

Observe que se um número é totalmente divisível sem restante, o resultado do módulo é 0. Podemos usar isso para testar números pares, pois se um número nódulo 2 for igual a 0, isso significa que é um número par!

Volte para o **for**!

Exemplo 2

Vamos imprimir apenas os números pares dessa lista!

In [4]:
```
for num in l:
    if num % 2 == 0:
        print(num)
```

```
2
4
6
8
10
```

Nós também poderíamos usar um else lá, também:

In [5]:
```
for num in l:
    if num % 2 == 0:
        print(num)
```

```
    else:
        print('Número ímpar')
```

```
Número ímpar
2
Número ímpar
4
Número ímpar
6
Número ímpar
8
Número ímpar
10
```

Exemplo 3

Outra idéia comum durante um **for** é manter algum tipo de contagem durante os vários loops. Por exemplo, vamos criar um loop for que resume a lista:

In [6]:
```
list_sum = 0

for num in l:
    list_sum = list_sum + num

print(list_sum)
```

```
55
```

Ótimo! Leia sobre a célula acima e certifique-se de entender completamente o que está acontecendo. Também poderíamos ter implementado um += para a adição. Por exemplo:

In [7]:
```
list_sum = 0

for num in l:
    list_sum += num

print(list_sum)
```

```
55
```

Exemplo 4

Nós usamos para loops com listas, e as strings? Lembre-se de que as strings são uma seqüência, então, quando iteramos através delas, estaremos acessando cada item nessa seqüência de caracteres.

In [8]:
```
for letter in 'This is a string.':
    print(letter)
```

```
T
h
i
s

i
s

a

s
```

```
t
r
i
n
g
.
```

Example 5

E com tuplas?

In [16]:
```
tup = (1,2,3,4,5)

for t in tup:
    print t
```

```
1
2
3
4
5
```

Exemplo 6

As Tuplas têm uma qualidade especial quando se trata de **fors**. Se você está iterando através de uma seqüência que contém tuplas, o item pode realmente ser a própria tupla, este é um exemplo de *desembalagem de tuplas* . Durante o **for**, estaremos desembalando a tupla dentro de uma seqüência e podemos acessar os itens individuais dentro dessa tupla!

In [9]:
```
l = [(2,4),(6,8),(10,12)]
```

In [11]:
```
for tup in l:
    print(tup)
```

```
(2, 4)
(6, 8)
(10, 12)
```

In [13]:
```
# Agora desembalando
for (t1,t2) in l:
    print(t1)
```

```
2
6
10
```

Legal! Com as tuplas em uma seqüência, podemos acessar os itens dentro deles por meio de desembalagem! A razão pela qual isso é importante é porque muitos objetos entregarão seus iterables através de tuplas. Vamos começar a explorar a iteração através de Dictionaries para explorar isso ainda mais!

Exemplo 7

In [14]:
```
d = {'k1':1,'k2':2,'k3':3}
```

In [15]:
```
for item in d:
    print(item)
```

```
k3
k2
k1
```

Observe como isso produz apenas chaves. Então, como podemos obter os valores? Ou as chaves e os valores?

items()

Você deve usar .items() para iterar através das chaves e valores de um dicionário. Por exemplo:

In [11]:
```
# For Python 3
for k,v in d.items():
    print(k)
    print(v)
```

```
k3
3
k2
2
k1
1
```

Conclusão

Aprendemos a usar para loops para iterar através de tuplas, listas, strings e dicionários. Será uma ferramenta importante para nós, portanto, certifique-se de conhecê-lo bem e compreende os exemplos acima.

Mais recursos

ASIMOV

While

A instrução **while** em Python é uma das formas mais gerais de executar iterações. Uma instrução **while** executará repetidamente uma única declaração ou grupo de instruções, desde que a condição seja verdadeira. A razão pela qual é chamado de "loop" é porque as instruções de código são roteadas repetidamente até que a condição não seja mais atendida.

O formato geral de um loop while é:

while teste: declaração de código else: declarações de código final

Olhemos alguns simples do while em ação.

In [1]:
```
x = 0

while x < 10:
    print('x is currently: ', x)
    print(' x is still less than 10, adding 1 to x')
    x += 1
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
x is currently:  1
 x is still less than 10, adding 1 to x
x is currently:  2
 x is still less than 10, adding 1 to x
x is currently:  3
 x is still less than 10, adding 1 to x
x is currently:  4
 x is still less than 10, adding 1 to x
x is currently:  5
 x is still less than 10, adding 1 to x
x is currently:  6
 x is still less than 10, adding 1 to x
x is currently:  7
 x is still less than 10, adding 1 to x
x is currently:  8
 x is still less than 10, adding 1 to x
x is currently:  9
 x is still less than 10, adding 1 to x
```

Observe quantas vezes as declarações de impressão ocorreram e como o while continuou até a condição True deixasse de ser verdadeira, que ocorreu após x == 10. É importante notar que, uma vez que isso ocorreu, o código parou. Vamos ver como podemos adicionar uma outra afirmação:

In [2]:
```
x = 0

while x < 10:
    print('x is currently: ',x)
```

```
    print(' x is still less than 10, adding 1 to x')
    x+=1

else:
    print('All Done!')
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
x is currently:  1
 x is still less than 10, adding 1 to x
x is currently:  2
 x is still less than 10, adding 1 to x
x is currently:  3
 x is still less than 10, adding 1 to x
x is currently:  4
 x is still less than 10, adding 1 to x
x is currently:  5
 x is still less than 10, adding 1 to x
x is currently:  6
 x is still less than 10, adding 1 to x
x is currently:  7
 x is still less than 10, adding 1 to x
x is currently:  8
 x is still less than 10, adding 1 to x
x is currently:  9
 x is still less than 10, adding 1 to x
All Done!
```

break, continue, pass

Podemos usar as declarações break, continue e pass em nossos loops para adicionar funcionalidades adicionais para vários casos. As três declarações são definidas por:

break: Para o loop continue: Vai para o próximo loop pass: Não faz nada
Pensando nas declarações **break** e **continue** , o formato geral do loop while se parece com isto:

while test: código if test: break if test: continue else:

Vamos ver alguns exemplos!

In [3]:
```
x = 0

while x < 10:
    print('x is currently: ', x)
    print(' x is still less than 10, adding 1 to x')
    x+=1
    if x ==3:
        print('x==3')
    else:
        print('continuing...')
        continue
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
continuing...
x is currently:  1
 x is still less than 10, adding 1 to x
continuing...
x is currently:  2
 x is still less than 10, adding 1 to x
x==3
```

```
x is currently:  3
 x is still less than 10, adding 1 to x
continuing...
x is currently:  4
 x is still less than 10, adding 1 to x
continuing...
x is currently:  5
 x is still less than 10, adding 1 to x
continuing...
x is currently:  6
 x is still less than 10, adding 1 to x
continuing...
x is currently:  7
 x is still less than 10, adding 1 to x
continuing...
x is currently:  8
 x is still less than 10, adding 1 to x
continuing...
x is currently:  9
 x is still less than 10, adding 1 to x
continuing...
```

Observe como temos uma declaração impressa quando x == 3 e continuamos imprimindo enquanto continuamos através do while. Vamos fazer uma pausa uma vez x == 3 e ver se o resultado faz sentido:

In [4]:

```
x = 0

while x < 10:
    print('x is currently: ',x)
    print(' x is still less than 10, adding 1 to x')
    x+=1
    if x ==3:
        print('Breaking because x==3')
        break
    else:
        print('continuing...')
        continue
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
continuing...
x is currently:  1
 x is still less than 10, adding 1 to x
continuing...
x is currently:  2
 x is still less than 10, adding 1 to x
Breaking because x==3
```

Observe a declaração else não foi alcançada e a continuação nunca foi impressa! Após esses exemplos breves e simples, você deve se sentir confortável ao usar as instruções em seu código.

Uma observação importante! É possível criar um ciclo de execução infinita com instruções while. Por exemplo:

In [ ]:

```
# NÃO RODE ESTE CÓDIGO!
while True:
    print('Uh Oh infinite Loop!')
```

ASIMOV

Compreensão em listas

Além das operações de sequência e métodos de lista, o Python inclui uma operação mais avançada chamada de compreensão de lista.

As compreensões de lista nos permitem construir listas usando uma notação diferente. Você pode pensar nisso essencialmente como um loop construído dentro de colchetes. Um exemplo simples:

Exemplo 1

In [1]:
```
# Pega todas as letras em uma string
lst = [x for x in 'word']
```

In [2]:
```
# Checa
lst
```

Out[2]:
```
['w', 'o', 'r', 'd']
```

Esta é a idéia básica de uma lista de compreensão. Se você estiver familiarizado com a notação matemática, esse formato deve se sentir familiar, por exemplo: x ^ 2: x em {0,1,2 ... 10}

Vejamos mais alguns exemplos de compreensões de lista em Python:

Exemplo 2

In [3]:
```
# Eleva o quadrado itens no range e o transformam em lista
lst = [x**2 for x in range(0,11)]
```

In [2]:
```
lst
```

Out[2]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Exemplo 3

Vamos ver como adicionar usando if:

In [4]:
```
# Cria uma lista de números pares, utilizando para isso o if
lst = [x for x in range(11) if x % 2 == 0]
```

In [6]:
```
lst
```

Out[6]: [0, 2, 4, 6, 8, 10]

Exemplo 4

Também pode fazer operações aritméticas mais complicadas:

In [6]:
```
# Converte Celsius para Fahrenheit
celsius = [0,10,20.1,34.5]

fahrenheit = [ ((float(9)/5)*temp + 32) for temp in celsius]

fahrenheit
```

Out[6]: [32.0, 50.0, 68.18, 94.1]

Exemplo 5

Também podemos realizar compreensões de lista aninhadas, por exemplo:

In [8]:
```
lst = [ x**2 for x in [x**2 for x in range(11)]]
lst
```

Out[8]: [0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]

Mais tarde, no curso, aprenderemos sobre compreensões do gerador. Após esta palestra, você deve se sentir confortável em ler e escrever compreensões de lista básica.

ASIMOV

Funções

Introdução às Funções

Esta aula consistirá em explicar o que é uma função em Python e como criar uma. As funções serão um dos nossos principais blocos de construção quando construiremos quantidades maiores e maiores de código para resolver problemas.

Então, o que é uma função?

Formalmente, uma função é um dispositivo útil que agrupa um conjunto de instruções para que elas possam ser executadas mais de uma vez. Eles também podem nos permitir especificar parâmetros que possam servir como entradas para as funções.

Em um nível mais fundamental, as funções nos permitem não ter que repetidamente escrever o mesmo código repetidas vezes. Se você lembrar de volta às lições em strings e listas, lembre-se de que usamos uma função len() para obter o comprimento de uma string. Uma vez que verificar o comprimento de uma seqüência é uma tarefa comum, você provavelmente vai querer escrever uma função que pode fazer isso repetidamente.

As funções serão um dos níveis mais básicos de código de reutilização em Python, e também nos permitirá começar a pensar no design do programa (mergulhaemos muito mais nas idéias de design quando aprendemos sobre programação orientada a objetos).

def

Vamos ver como construir a sintaxe de uma função em Python. Ela tem a seguinte forma:

In [3]:
```
def name_of_function(arg1,arg2):
    '''
    A documentação da função ficará aqui
    '''
    # Faça coisas aqui
    return # retorne o resultado desejado aqui
```

Começamos com def, seguido do nome da função. Tente manter os nomes relevantes, por exemplo len() é um bom nome para uma função length(). Também tenha cuidado com os nomes, você não gostaria de chamar uma função do mesmo nome que uma função interna em Python (como len).

Em seguida, venha um par de parênteses com vários argumentos separados por uma vírgula. Esses argumentos são as entradas para sua função. Você poderá usar essas entradas em sua função e fazer referência a elas. Depois disso, você coloca dois pontos.

Agora, aqui é o passo importante, você deve identar para começar o código dentro de sua função corretamente. Python faz uso de *espaço em branco* para organizar o código. Muitas outras linguagens de programação não fazem isso, então tenha isso em mente.

Em seguida, você verá o doc-string, é aqui que você escreve uma descrição básica da função. Usando iPython e iPython Notebooks, você será capaz de ler estes documentos pressionando Shift + Tab após um nome de função. Documentações não são necessárias para funções simples, mas é uma boa prática colocá-las para que você ou outras pessoas possam facilmente entender o código que você escreve.

Depois de tudo isso, você começa a escrever o código que deseja executar.

A melhor maneira de aprender funções é através de exemplos. Então, vamos tentar passar por exemplos que se relacionam com os vários objetos e estruturas de dados que aprendemos antes.

Exemplo 1: Uma função simples de 'Olá'

In [5]:
```
def say_hello():
    print('hello')
```

Chame a função

In [6]:
```
say_hello()
```

```
hello
```

Exemplo 2: Uma função de saudação simples

Vamos escrever uma função que cumprimenta pessoas com seu nome.

In [9]:
```
def greeting(name):
    print('Hello, %s' %name)
```

In [10]:
```
greeting('Rodrigo')
```

```
Hello, Rodrigo
```

Usando o return

Vamos ver um exemplo que usa uma declaração de retorno. Return permite uma função para *retornar* um resultado que pode ser armazenado como uma variável, ou usado de qualquer maneira que um usuário deseje.

Exemplo 3: função de adição

In [11]:
```
def add_num(num1,num2):
    return num1+num2
```

In [12]:
```
add_num(4,5)
```

Out[12]: 9

In [13]:
```
result = add_num(4,5)
```

In [15]:
```
print(result)
```

```
9
```

O que acontece se inserimos duas strings?

In [12]:
```
print add_num('one','two')
```

```
onetwo
```

Note que, porque não declaramos tipos de variáveis em Python, esta função pode ser usada para adicionar números ou seqüências em conjunto! Mais tarde, aprenderemos sobre a adição de verificações para garantir que um usuário coloque os argumentos corretos em uma função.

Vamos também começar a usar as instruções *break*, *continue* e *pass* no nosso código. Nós apresentamos estes durante a palestra de tempo.

Finalmente, vamos passar por um exemplo completo de criar uma função para verificar se um número é primo (um exercício de entrevista comum).

Nós sabemos que um número é primordial se esse número é apenas divisível em 1 e em si mesmo. Vamos escrever a nossa primeira versão da função para verificar todos os números de 1 a N e executar verificações de módulo.

In [17]:
```
def is_prime(num):
    '''
    Método para checar se é primo
    '''
    for n in range(2,num):
        if num % n == 0:
            print('Não primo')
            break
    else: # Se o módulo nunca for zero, é primo
        print('Primo')
```

In [18]:
```
is_prime(16)
```

```
Não primo
```

Observe como quebramos o código após a declaração de impressão! Na verdade, podemos melhorar isso ao verificar somente a raiz quadrada do número-alvo, também podemos ignorar todos os números pares depois de verificar 2. Também mudaremos para retornar um valor booleano para obter um exemplo de usar declarações de retorno:

In [19]:
```
import math

def is_prime(num):
    '''
    Melhor método para checar primos
    '''
    if num % 2 == 0 and num > 2:
```

```
        return False
    for i in range(3, int(math.sqrt(num)) + 1, 2):
        if num % i == 0:
            return False
    return True
```

In [20]:
```
is_prime(14)
```

Out[20]:
```
False
```

Ótimo! Você deve agora ter uma compreensão básica sobre como criar suas próprias funções para salvar-se de escrever repetidamente o código!

ASIMOV

---

map ()

map () é uma função que leva em dois argumentos: uma função e uma seqüência iterable. Na forma: map(função, sequência) O primeiro argumento é o nome de uma função e a segunda uma seqüência (por exemplo, uma lista). map() aplica a função a todos os elementos da seqüência. Ele retorna uma nova lista com os elementos alterados por função.

Quando fomos sobre a compreensão da lista, criamos uma pequena expressão para converter Fahrenheit a Celsius. Vamos fazer o mesmo aqui, mas usando map. Começaremos com duas funções:

In [8]:
```
def fahrenheit(T):
    return ((float(9)/5)*T + 32)
def celsius(T):
    return (float(5)/9)*(T-32)

temp = [0, 22.5, 40,100]
```

Agora vamos ver o map() em ação:

In [12]:
```
F_temps = list(map(fahrenheit, temp))

# Mostra
F_temps
```

Out[12]:
```
[32.0, 72.5, 104.0, 212.0]
```

In [13]:
```
# Converte devolta
list(map(celsius, F_temps))
```

Out[13]:
```
[0.0, 22.5, 40.0, 100.0]
```

No exemplo acima, não usamos uma expressão lambda. Ao usar lambda, não teríamos que definir e nomear as funções fahrenheit() e celsius().

In [14]:
```
list(map(lambda x: (5.0/9)*(x - 32), F_temps))
```

Out[14]:
```
[0.0, 22.5, 40.0, 100.0]
```

Ótimo! Nós obtivemos o mesmo resultado! O uso do map() é muito mais comumente usado com expressões lambda, já que todo o propósito do map() é economizar esforço ao ter que criar manual para loops.

map() pode ser aplicado a mais de um iterable. Os iteráveis devem ter o mesmo comprimento.

Por exemplo, se estamos trabalhando com duas listas-map() aplicará sua função lambda aos elementos das listas de argumentos, ou seja, aplica-se primeiro aos elementos com o índice 0, e depois aos elementos com o 1º índice até o que o índice N seja alcançado.

Por exemplo, mapeamos uma expressão lambda para duas listas:

In [16]:
```
a = [1,2,3,4]
b = [5,6,7,8]
c = [9,10,11,12]

list(map(lambda x,y:x+y,a,b))
```

Out[16]:
```
[6, 8, 10, 12]
```

In [18]:
```
list(map(lambda x,y,z:x+y+z, a,b,c))
```

Out[18]:
```
[15, 18, 21, 24]
```

Podemos ver no exemplo acima que o parâmetro x obtém seus valores da lista a, enquanto y obtém seus valores de b e z da lista c. Vá em frente e teste com seu próprio exemplo para se certificar de que compreende o mapeamento para mais do que um iterable.

Bom trabalho! Você deve agora ter uma compreensão básica da função map ().

ASIMOV

Arquivos

O Python usa objetos de arquivo para interagir com arquivos externos em seu computador. Esses objetos de arquivo podem ser qualquer tipo de arquivo que você tenha em seu computador, seja um arquivo de áudio, um arquivo de texto, e-mails, documentos do Excel, etc. Nota: provavelmente você precisará instalar certas bibliotecas ou módulos para interagir com esses vários tipos de arquivo, mas eles estão facilmente disponíveis. (Vamos abordar o download de módulos mais tarde no curso).

O Python possui uma função aberta aberta que nos permite abrir e utilizar métodos básicos com arquivos. Primeiro, precisamos de um arquivo. Usaremos uma mágica do iPython para criar um arquivo de texto!

Escrevendo um arquivo com iPython

In [1]:
```
%%writefile test.txt
Hello, this is a quick test file
```

```
Writing test.txt
```

Abrindo um arquivo com Python

Podemos abrir um arquivo com a função open(). A função open também possui argumentos (também chamados de parâmetros). Vamos ver como isso é usado:

In [2]:
```
# Abre um arquivo txt já existente
my_file = open('test.txt')
```

In [3]:
```
# Agora podemos ler o arquivo
my_file.read()
```

Out[3]:
```
'Hello, this is a quick test file'
```

In [4]:
```
# Mas o que acontece se tentarmos lê-lo novamente?
my_file.read()
```

Out[4]:
```
''
```

Isso acontece porque você pode imaginar que o "cursor" de leitura esteja no final do arquivo depois de ter lido. Portanto, não há nada a ler. Podemos redefinir o "cursor" assim:

In [8]:
```
# Procure o início do arquivo (índice 0)
```

```
my_file.seek(0)
```

Out[8]: 0

In [6]:
```
# Lê novamente
my_file.read()
```

Out[6]: 'Hello, this is a quick test file'

Para não ter que reiniciar todas as vezes, também podemos usar o método readlines. Tenha cuidado com arquivos grandes, já que tudo será mantido na memória. Nós aprenderemos como iterar sobre arquivos grandes mais tarde no curso.

In [9]:
```
# Readlines retorna uma lista das linhas no arquivo.
my_file.readlines()
```

Out[9]: ['Hello, this is a quick test file']

Escrevendo um arquivo

Por padrão, usando a função open() só nos permitirá ler o arquivo, precisamos passar o argumento 'w' para escrever sobre o arquivo. Por exemplo:

In [39]:
```
# Adiciona um segundo argumento à função, 'w', que significa escrita
my_file = open('test.txt','w+')
```

In [40]:
```
# Escreve no arquivo
my_file.write('This is a new line')
```

In [43]:
```
# Lê o arquivo
my_file.read()
```

Out[43]: 'This is a new line'

Iterando através de um arquivo

Vamos dar uma pequena olhada sobre como iterar através do arquivo. Primeiro vamos criar um novo arquivo de texto:

In [44]:
```
%%writefile test.txt
First Line
Second Line
```

```
Overwriting test.txt
```

Agora podemos usar um pouco de fluxo para dizer o programa para através de cada linha do arquivo e fazer algo:

In [45]:
```
for line in open('test.txt'):
    print(line)
```

```
First Line

Second Line
```

Não se preocupe em entender completamente isso ainda, pois os laços serão abordados em breve. Mas vamos quebrar o que fizemos acima. Nós dissemos que para cada linha neste arquivo de texto, vá em frente e imprima essa linha. É importante notar algumas coisas aqui:

1.) Poderíamos ter chamado o objeto 'linha' qualquer coisa (veja o exemplo abaixo). 2.) Ao não chamar .read() no arquivo, o arquivo de texto inteiro não foi armazenado na memória.
3.) Observe o recuo na segunda linha para impressão. Este espaço em branco é necessário em Python.

Aprenderemos muito mais sobre isso mais tarde.

In [46]:
```
for asdf in open('test.txt'):
    print(asdf)
```

```
First Line

Second Line
```

ASIMOV

---

reduce()

Muitas vezes, os alunos têm dificuldade em entender reduce() por isso, preste muita atenção a esta palestra. A função reduce(função, sequência) aplica continuamente a função à sequência. Em seguida, ele retorna um único valor.

Se seq = [s1, s2, s3, ..., sn], a redução de chamada (função, seqüência) funciona assim:

- No início, os dois primeiros elementos de seq serão aplicados à função, isto é, func(s1, s2)
- A lista em que a reduce() funciona parece assim: [função (s1, s2), s3, ..., sn]
- No próximo passo, a função será aplicada no resultado anterior e no terceiro elemento da lista, ou seja, função(função (s1, s2), s3)
- A lista parece agora: [função (função (s1, s2), s3), ..., sn]
- Continua assim até apenas um elemento é deixado e retorna esse elemento como resultado de reduzir ()

Vamos ver um exemplo:

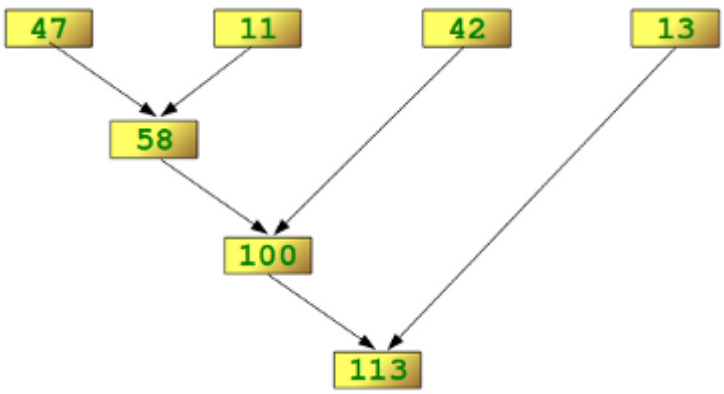
In [16]:
```
lst =[47,11,42,13]
reduce(lambda x,y: x+y,lst)
```

Out[16]:
```
113
```

Vamos ver o diagrama para entender melhor.

In [9]:
```
from IPython.display import Image
Image('http://www.python-course.eu/images/reduce_diagram.png')
```

Out[9]:



Observe como continuamos reduzindo a seqüência até obter um único valor final. Vamos ver outro exemplo:

In [20]:
```
#Find the maximum of a sequence (This already exists as max())
```

```
max_find = lambda a,b: a if (a > b) else b
```

In [21]:
```
#Find max
reduce(max_find,lst)
```

Out[21]: 47

Hopefully you can see how useful reduce can be in various situations. Keep it in mind as you think about your code projects!