Python

Curso de Introdução à Programação de Jogos em Python

OFERTA PILOTO 2017

Material de estudo

Ministério da Educação

SEB

LabTIME

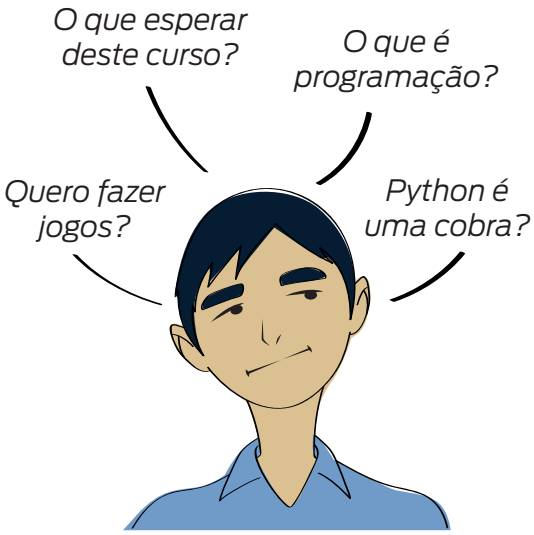
UFG

SUMÁRIO

Introdução

Neste curso aprenderemos noções básicas de programação e de desenvolvimento de jogos usando Python. Mas antes, precisamos saber de fato de que se trata e o porquê de seu uso neste curso.

Começaremos pensando a respeito do que queremos deste curso. Queremos aprender a fazer jogos, certo? Quem vai executar estes jogos? Bem, os jogos serão executados pelo computador. Até aqui tudo bem, mas como nos comunicaremos com ele? Como o faremos entender o que queremos? Através de uma linguagem de programação. Python é a linguagem que usaremos para este fim.



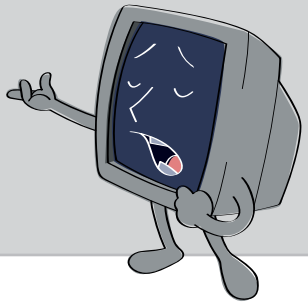
A linguagem de máquina

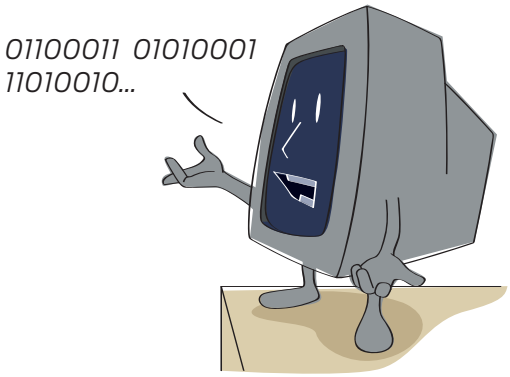
Na realidade, o computador não entende a linguagem Python, mas Python não é apenas uma linguagem, é, também, um **programa intérprete**. Este programa faz a tradução da linguagem Python para a linguagem de máquina, que é, na verdade, a linguagem "falada" pelos computadores. Com isso você deve estar se perguntando: então por que não usar diretamente a linguagem de máquina e evitar intermediários? Porque a linguagem de máquina consiste de uma série de instruções e dados representados em código binário (compostos por 0's e 1's) de difícil compreensão para nós, humanos. Por isso usamos linguagens como Python, que são mais compreensíveis e cujas instruções podem ser traduzidas à linguagem de máquina.

>>>

Programa Intérprete

Traduz as instruções do código de uma linguagem de programação (Python, por exemplo) para a linguagem de máquina (representada por código binário, 0 e 1) entendida pelos computadores.



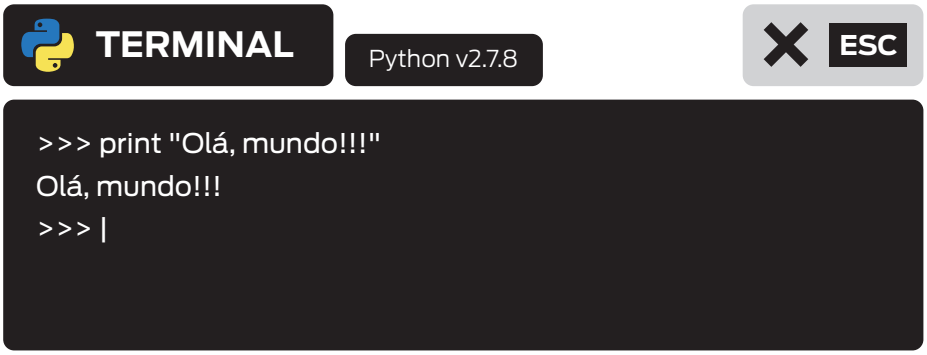


Por que Python?

Python é considerada uma das linguagens mais fáceis de aprender que existe e seu uso vem crescendo a ponto de ser uma das mais populares. Infelizmente para nós, brasileiros, a maioria das linguagens importantes de programação está baseada no inglês e isto também se aplica à linguagem Python. Como um exemplo, temos o comando *print*. **"Print"** em inglês significa **"imprima"**. Logo, este comando faz com que seja impresso na tela o que mandarmos imprimir. Uma das razões para esta facilidade é o seu modo interativo, que nos permite explorar a linguagem e testar pequenos códigos (programas). A seguir, veremos como executar instruções usando o modo interativo.

Modo interativo

Python oferece uma forma de executar códigos interativamente. No próprio jogo podemos abrir um terminal para executar pequenos códigos de forma interativa. As instruções devem ser colocadas após o símbolo ">>>" e para executá-las pressionamos a tecla ENTER. Faça um pequeno teste, coloque a instrução abaixo no terminal e pressione ENTER.



Perceba que na tela, logo abaixo da instrução, será impresso "Olá, mundo!!!".

Mensagens de erros

Se você for atento, terá percebido que não foi colocado ponto no final da instrução. Se você colocar um ponto no final da linha de código e repetir todo o procedimento, você terá:

>>> print "Olá, mundo!!!".

Após pressionarmos ENTER, veremos que aparece uma mensagem de erro parecida com a seguinte:

SyntaxError: invalid syntax

O que isto significa? Significa que o Python não conseguiu executar a instrução devido a algum problema. Em particular, ocorreu um erro de sintaxe ("syntax", em inglês). Sintaxe

é o conjunto de regras que regem a escrita de uma linguagem de programação. Em Python não é permitido que uma instrução termine com um ponto (“.”), por isso ocorre a mensagem de erro. Certamente você notou que a mensagem está em inglês. Isto é um problema, mas com a prática você irá se acostumando e compreendendo essas mensagens.

Existem outros erros de sintaxe e, inclusive, outros tipos de erros que serão vistos à medida que formos programando. Se muitas destas mensagens aparecerem em seu código, não se desespere (bem-vindo ao mundo da programação!). Primeiro analise o que foi escrito, use a mensagem de erro para tentar entender o problema e reescreva as instruções.

Se tivéssemos colocado um ponto e vírgula (“;”) em vez do ponto, não teria aparecido a mensagem de erro. Isto porque a sintaxe de Python permite que uma instrução termine com “;”.

Python 2.x ≠ Python 3.x

No momento, são atualizadas duas versões de Python, a 2 (com suas subversões) e a 3. Por que isto? A versão 3 é uma versão melhorada mas incompatível com versões anteriores. Então por que usar a versão 2? Porque existem (ainda) muitas ferramentas escritas para esta versão que não foram adaptadas para a versão 3.

Estas versões apresentam algumas diferenças de sintaxe.

Vejamos como ficaria o código para imprimir “Olá, mundo!!!!” nas duas versões:

Python 2.x:

```
>>>print “Olá, mundo!!!”
Olá, mundo!!!
```

Python 3.x:

```
print (“Olá, mundo!!!”)
Olá, mundo!!!
```

Se não usarmos os parênteses na versão 3 aparecerá um erro de sintaxe. Existem outras diferenças, mas não é o objetivo deste curso aprofundarmos nisto.

Lembre-se que a versão que adotaremos para o curso é a 2.7.

Valores

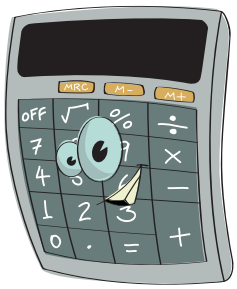
Em nosso primeiro código, aquele que imprimiu "Olá, mundo!!!", tivemos o primeiro exemplo de valor. Nele, usamos o comando *print* e o valor "Olá, mundo!!!". Esse é um valor do tipo **string** (alfanumérico) – daremos mais detalhes sobre este tipo quando falarmos dos tipos de valores.

Existem valores de outros tipos, como números (inteiros e floats), listas, tuplas, etc. Sem falar que podemos criar os nossos próprios tipos de valores. Mas deixaremos isto para o final deste tutorial, onde veremos que os valores, na verdade, são objetos em Python (não se preocupe com isto, por enquanto).

Bem como numa calculadora, podemos realizar algumas operações com valores para obter novos valores. Façamos um pequeno teste no terminal do jogo:

TERMINAL Python v2.7.8 ESC

```
>>> 4 + 5
9
>>>
```



Depois de digitar a primeira linha do código e pressionar ENTER, verá que aparece o valor 9. Aqui foi feita a soma do valor inteiro 4 com o valor inteiro 5, obtendo o valor inteiro 9. Os tipos dos valores nos determinam as operações que podemos realizar. Neste exemplo, realizamos a soma de dois números.

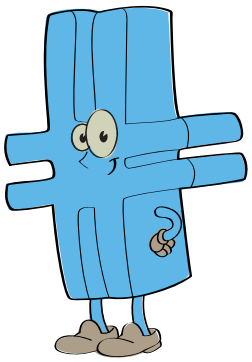
Variáveis

Pode ocorrer, algumas vezes, de querermos trabalhar com algum valor usado antes. Será que existe uma forma de guardar um valor para uso posterior? Existe! Para isto usamos o que chamamos de "variável".

Variável é um nome associado a um valor que pretende ser usado posteriormente em um código. Em Python, para atribuir um valor a uma variável usamos o operador "=". O que estiver à esquerda será o nome da variável, e o que estiver à direita será o valor associado a ela. Exemplo:

TERMINAL Python v2.7.8 ESC

```
>>> x = 4              # a variável x guardará o valor 4
>>> print x            # imprime 4
4
>>>
```

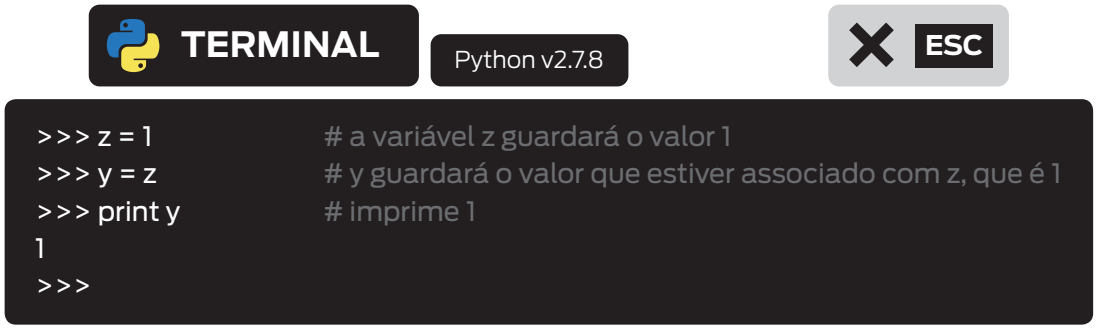


NOTA: O símbolo "#" indica que será introduzido um comentário à sua direita. Comentários são úteis para dar alguma informação a quem lê o código (inclusive você mesmo). Python, simplesmente, ignora os comentários na hora de executar um código.

Acabamos de dizer que do lado direito de "=" deverá haver um valor, mas nem sempre esse valor estará de forma explícita. Vejamos alguns exemplos:

Exemplo 1:

Exemplo 2:



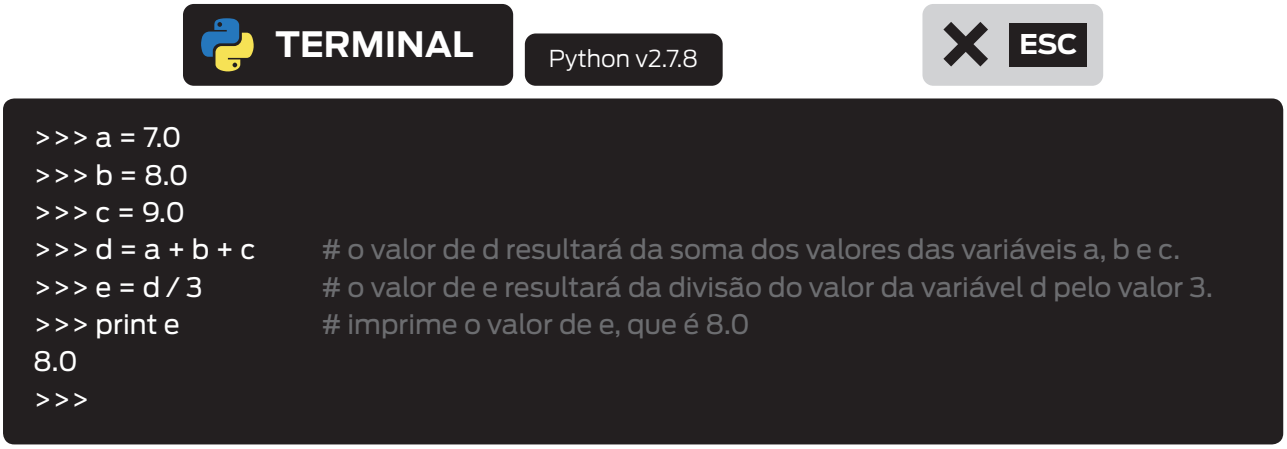
Neste último exemplo, se não existisse a primeira linha (z = 1), teria aparecido uma mensagem de erro, pois z não estaria definida (não guardaria nenhum valor). Veja:

Dizemos que uma variável está definida se ela guardar algum valor, ex: x = 5. O tipo de uma variável é o tipo do valor que ela guarda. No exemplo, temos que x (variável) é do tipo inteiro, pois 5 (valor) é do tipo inteiro.

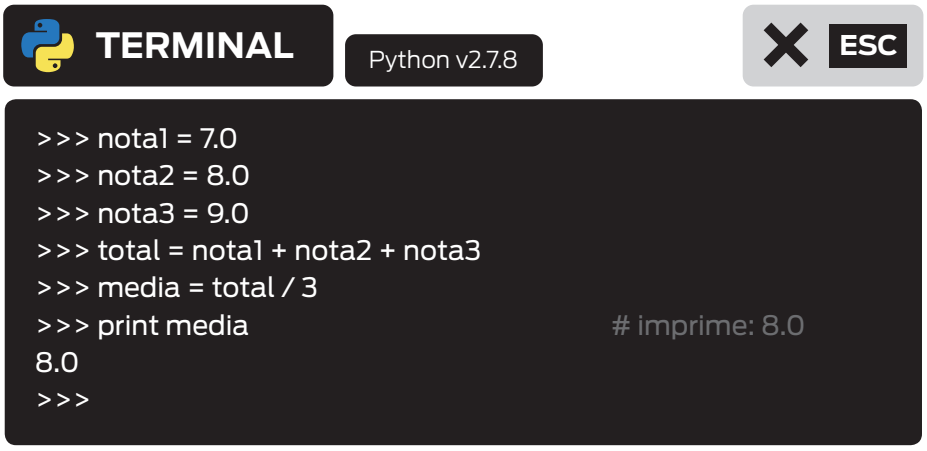
É bom escolher nomes pequenos e significativos para as variáveis. Em Python, nomes de variáveis podem conter tanto letras quanto números (e até o símbolo *underline* "_"), mas devem começar com uma letra ou "_". Espaços em branco e acentos não são permitidos na versão que estamos usando. Python diferencia as variáveis escritas com letras minúsculas das escritas com letras maiúsculas. Exemplo:

Abaixo temos dois códigos que fazem a mesma tarefa, porém usando diferentes nomes de variáveis. Perceba como nomes de variáveis adequados nos ajudam a entender o que está sendo feito.

Código 1:



Código 2:



Para praticar, recomendamos que você teste estes códigos no terminal do jogo.

Outro cuidado que devemos ter ao escolher o nome de uma variável é com as palavras da linguagem reservadas para usos especiais. Por exemplo, *for* é uma palavra reservada de Python, não podemos escolhê-la como nome de variável. Se fizermos: for = 3, aparecerá um erro, pois *for* é um comando para gerar repetições, como será visto mais adiante.

>>>

Lista de algumas palavras com uso especial em Python:

- for, print, and, del, from, while, elif, or, else, if, import, class, in, is, return, def.

Tipos de Valores (e variáveis)

Acima foi falado, brevemente, sobre variáveis e valores. Agora veremos seus principais tipos.

123 8.6 Valores numéricos

Valores numéricos? Bem...! São os números. Variáveis numéricas são aquelas que guardam valores numéricos. Trabalharemos com dois tipos de valores numéricos, os **inteiros** (*int*) e os números em ponto flutuante, que chamaremos de *floats* ou **decimais**. Conhecemos bem os inteiros, ex: 5. Já os *floats* são números que contêm uma parte decimal, ex: 3.2.

NOTA: Para representar números do tipo *float* usamos o ponto e não a vírgula.
Ex: 2.3 ---> Certo
2,3 ---> Errado

Operações com valores numéricos:

soma (+):

diferença (-):

multiplicação (*):

divisão (/):

divisão inteira (//):

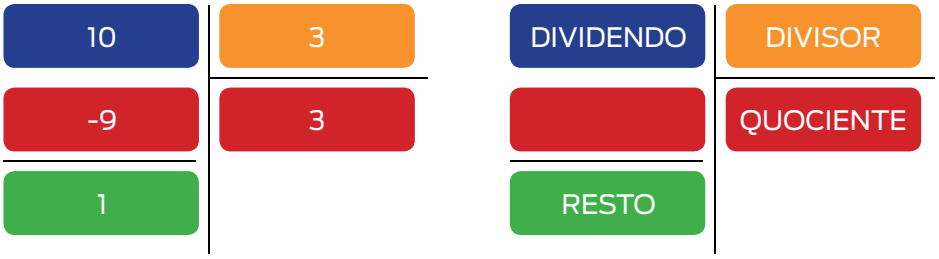
resto (%):

potenciação (**):

Os operadores **+**, **-**, ***** e **/** são familiares, mas como se comportam os operadores **//**, **%** e ****?

O operador **//** é de divisão inteira. Para compreendermos melhor isto, vamos relembrar dos velhos tempos quando aprendíamos a dividir. Lembra-se da seguinte regra:

dividendo = divisor * quociente + resto



Quando dividimos 10 por 3 obtemos 3 com resto 1 (10 = 3 * 3 + 1). Se fizermos 299 dividido por 100, obteremos 2 com resto 99 (299 = 100 * 2 + 99). O resultado da divisão inteira é o quociente. Logo, 10 // 3 = 3 e 299 // 100 = 2. Se o dividendo e o divisor forem, ambos, positivos ou negativos, podemos obter o resultado da divisão inteira, simplesmente, fazendo a divisão "normal" (/) e removendo a parte decimal. Ex: 10.0 / 3 = 3.33...; sem a parte decimal, obtemos 3. Aqui devemos ter um pouco de cuidado com a versão de Python que estivermos usando. Na versão 3, fazer 10 / 3 resultará em 3.33..., mas na versão 2, resultará em 3. Isto se deve ao fato de que, se usarmos o operador "/" com dois inteiros resultará em um inteiro (divisão inteira), mas se pelo menos um dos operandos for do tipo float, resultará em um número desse tipo, no caso 3.33...

O que dizer sobre a operação resto? Ela é realizada com o operador % em Python. Logo, 10 % 3 = 1 e 299 % 100 = 99. Note que o resto será sempre positivo (ou zero) e menor que o divisor (se este e o dividendo forem positivos).

O operador ** é a potenciação. Podemos calcular 2^3 fazendo 2**3, por exemplo.

Precedência de operadores aritméticos

Abramos o terminal do jogo e coloquemos o seguinte:

Qual você acha que seria a resposta?

Logo depois de pressionarmos ENTER obtemos 3.0 como resposta, por quê?

A resposta está em como Python escolhe a ordem das operações (precedência) para realizar os cálculos e evitar ambiguidades. A precedência em Python para operadores aritméticos é a mesma da matemática:

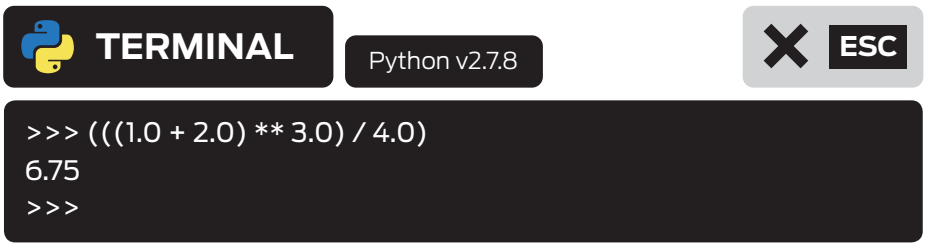
1º) **

2º) *, /, //, %

3º) +, -

No exemplo dado, podemos ver que ** tem maior precedência, logo o primeiro a ser calculado é a operação com este operador e seus operandos, ou seja, **2.0 ** 3.0** (resposta, 8.0). A expressão ficaria, então: **1.0 + 8.0 / 4.0**. Desta expressão, o operador que tem maior precedência é **/**, então calcula a divisão **8.0 / 4.0** (resposta, 2.0). Depois deste cálculo a expressão ficaria: **1.0 + 2.0**. Aí, sim, é efetuada a soma. Como resultado, obteremos **3.0**.

Mas como teríamos que fazer se quiséssemos que as operações fossem executadas na ordem em que aparecem? Neste caso, deveríamos usar parênteses, como é mostrado abaixo:



OU

Que resultará em 6.75. Com o uso dos parênteses, serão executadas as operações com os parênteses mais internos primeiro. No caso 1.0 + 2.0 (= 3.0) primeiro; 3.0 ** 3.0 (= 27.0) segundo e, finalmente, 27.0 / 4.0 (= 6.75).

Vejamos outro exemplo. Que tal fazer:

Isto resultará em 3. Como os operadores **%** e ***** têm a mesma precedência, Python executará o cálculo da esquerda para a direita. No caso 5 % 2 (= 1) primeiro e 1 * 3 (= 3) por último. Se trocarmos os operadores de lugar, teremos: 5 * 2 % 3 = 1 (5 * 2 = 10; 10 % 3 = 1). A potenciação é uma exceção a esta regra, o cálculo se faz da direita para a esquerda. Ex: 2 ** 1 ** 2 é igual a 2 e não 4.

Valores booleanos

Normalmente, o computador executa as instruções na ordem em que são colocadas. Mas é possível pedir ao computador que repita certas instruções ou que faça coisas que não estão necessariamente na linha seguinte do código. **Laços** e **condicionais**, que serão vistos mais adiante, estão entre estes tipos de instruções. Neste tipo de estrutura, devemos indicar as condições para que uma parte do código seja executada (uma ou repetidas vezes). Nestas estruturas é onde os valores booleanos se tornam importantes.

Existem apenas 2 valores booleanos, True e False, que significam "verdadeiro" e "falso", respectivamente. Variáveis que guardam qualquer destes valores são chamadas **variáveis booleanas**. Ex: *var = True*. Perceba que estes valores devem começar com letra maiúscula e não usam aspas.

True ---> valor do tipo booleano

"True" ---> valor do tipo string

Expressões booleanas

São expressões que resultam em valores booleanos. É hora de fazer um pequeno teste no terminal do jogo. Abra um terminal e coloque:

Após pressionar ENTER aparecerá *True*. Isto ocorre porque, comparando os valores **100** e **101**, **100** é menor do que **101**. Logo, essa expressão é igual a *True* (verdadeiro). Vejamos outro exemplo:

O que será impresso? Será impresso *False*. O que aconteceu aqui foi o seguinte, como 3 não é maior do que 5 a expressão 3 > 5 retorna *False*. O valor *False* foi guardado na variável *valBool*. Finalmente, mandamos imprimir o valor desta variável, imprimindo *False*.

Os operadores "**<**" e "**>**" são operadores de comparação. A seguir serão mostrados outros operadores deste tipo.

Operadores de comparação:

==: compara se dois valores são iguais.

!= ou <>: compara se dois valores são diferentes.

>: retorna True se o valor à esquerda for maior que o da direita, senão retorna False.

<: retorna True se o valor à esquerda for menor que o da direita, senão retorna False.

>=: retorna True se o valor à esquerda for maior ou igual ao da direita, senão retorna False.

<=: retorna True se o valor à esquerda for menor ou igual ao da direita, senão retorna False.

NOTA: É importante notar a diferença entre os operadores **"="** e **"=="**. O primeiro serve para dar algum valor a uma variável. O segundo compara dois valores e retorna um valor booleano. Abaixo um exemplo de como o computador interpreta as instruções.

x = 4 -----> guarde o valor 4 na variável x

x == 4 -----> a variável x tem o valor 4?

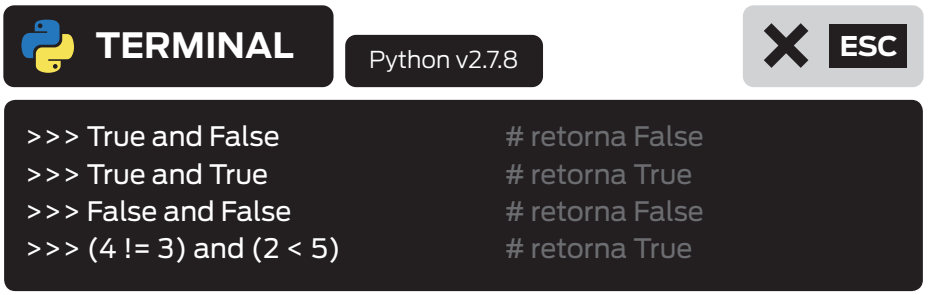
Neste último caso, x deve ter algum valor ou obteremos uma mensagem de erro.

Operadores lógicos:

Operadores lógicos operam com valores de tipo booleano (embora em Python possam operar com outros tipos de valores). Existem três operadores lógicos em Python:

not (negação): resultam no valor oposto.

and (e): retorna True, somente se os dois operandos forem True, senão retorna False.



or (ou): retorna False somente se os dois operandos forem False, senão retorna True.

Valores alfanuméricos (strings)

Valores alfanuméricos, que chamaremos mais frequentemente de *strings*, são compostos por sequências de caracteres que podem ser números, letras, espaço, símbolos. São delimitados por aspas, tanto duplas (“) como simples (‘) em Python. Já trabalhamos com um valor string neste tutorial. Você se lembra do “Olá, mundo!!!” em nosso primeiro código? Sim, ele é um valor do tipo *string.*

Abaixo criaremos algumas variáveis do tipo *string* como exemplo.

TERMINAL Python v2.7.8 **ESC**

```
>>> texto1 = ‘O rato roeu a roupa do rei de Roma.’
>>> texto2 = “3.23”
>>> texto3 = “”                    # duas aspas (simples ou duplas) sem espaço entre elas
>>> texto4 = ‘‘‘                   # três aspas (simples ou duplas) sem espaço entre elas
texto de
várias
linhas
’’’                                # final de texto4
```

Perceba que em texto2 temos uma *string* e não um número (*float*). A presença de aspas caracteriza o tipo alfanumérico. Por esta razão texto3 também é uma *string*, chamada *string* vazia. Já em texto4, temos uma *string* de várias linhas. Podemos conseguir isto usando três aspas no começo e no final da *string* (tanto aspas simples como duplas, mas sem misturá-las). Lembrando que para ir a uma nova linha é necessário apertar a tecla ENTER.

NOTA: Como fazer se quisermos usar aspas dentro de uma string? Da seguinte forma:

TERMINAL Python v2.7.8 ESC

```
>>> texto = 'Frase do Chapolin: "Nao contavam com a minha astucia"'
>>> print texto                    # imprime: Frase do Chapolin: "Não contavam com minha astúcia"
```

Perceba que as aspas mais internas são duplas, enquanto que as externas são simples. Também é possível realizar isto invertendo as aspas (mais internas simples e mais externas duplas).

Operações com strings:

Concatenação (+):

TERMINAL Python v2.7.8 ESC

```
>>> 'abc' + 'def'                 # retorna 'abcdef'
```

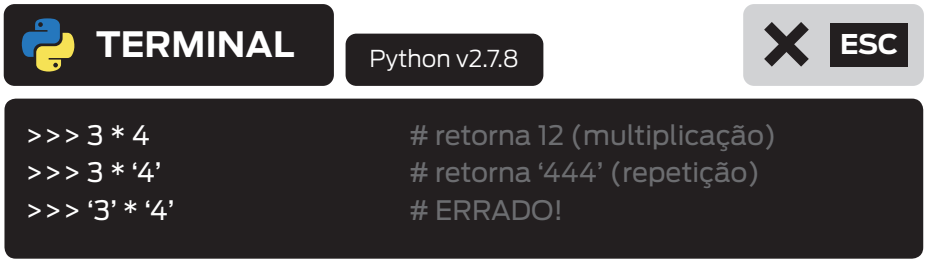
O operador **+** concatena (junta) strings. Tenha cuidado! Pois o símbolo do operador concatenação é o mesmo que o de soma, porém a soma é feita com tipos numéricos (inteiros ou decimais). Veja os seguintes exemplos:

TERMINAL Python v2.7.8 ESC

```
>>> 8 + 7                         # retorna 15 (soma)
>>> '8' + '7'                     # retorna '87' (concatenação)
>>> '8' + 7                       # ERRADO!
```

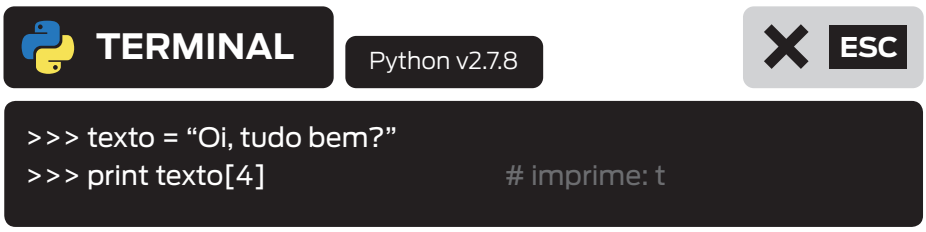
Repetição (*):

O operador * repete strings. Uma string será repetida tantas vezes quanto indicado pelo inteiro que a acompanha na operação. Novamente, cuidado! Com números, este operador funciona de forma diferente do que com strings. Veja este comportamento nos seguintes exemplos:



Indexação(string[pos]):

As strings são indexadas. Isto significa que podemos obter um caractere da string a partir de uma determinada posição.



Expliquemos um pouco como isto funciona. Na posição 0 temos o primeiro caractere ("O"), na posição 1 o segundo ("i"), e assim por diante. Na posição 4 temos o quinto caractere, que é "t". Perceba que espaços também são caracteres.

Fatiamento (string[i: f]):

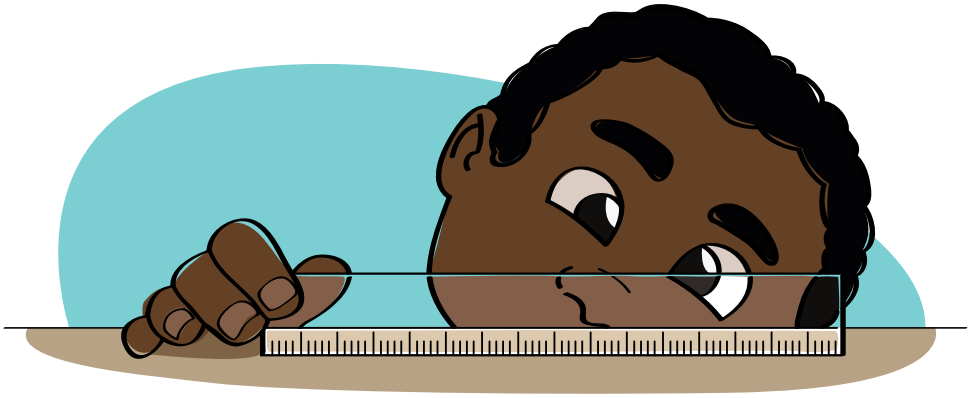
Com esta operação podemos obter uma string que seja uma cópia de parte de outra.

TERMINAL Python v2.7.8 ESC

```
>>> texto = “Oi, tudo bem?”
>>> print texto[4:12]                # imprime: tudo bem
```

Vejamos como funciona. Abaixo foi colocada a *string* entre espaços para melhor visualizar as posições de seus caracteres.

O	i	,		t	u	d	o		b	e	m	?	*string*
0	1	2	3	4	5	6	7	8	9	10	11	12	*posição dos caracteres*



Aqui é feita uma cópia de parte da string *texto*. São copiados os caracteres que estão presentes a partir da posição 4 (caractere “t”) até a posição anterior à 12 (caractere “m”). Perceba que a cópia será feita desde a posição que vem antes dos “:” (INCLUINDO o que estiver nessa posição) até a posição que vem depois dos “:” (EXCLUINDO o que estiver nessa posição).

NOTA: Também é possível fazer:

TERMINAL Python v2.7.8 ESC

```
>>> print texto[4:]                  # imprime: tudo bem?
```

Isto imprimirá uma cópia dos caracteres que estiverem a partir da posição 4 até o final da string *texto*.

Ou ainda:

TERMINAL Python v2.7.8 ESC

```
>>> print texto[:2]                    # imprime: Oi
```

Isto imprimirá uma cópia desde o começo até a posição 2 da *string* texto (excluindo o que estiver nessa posição).

Finalmente:

TERMINAL Python v2.7.8 ESC

```
>>> print texto[:]                     # imprime: Oi, tudo bem?
```

Isto imprimirá uma cópia da *string* texto.

[,] Listas

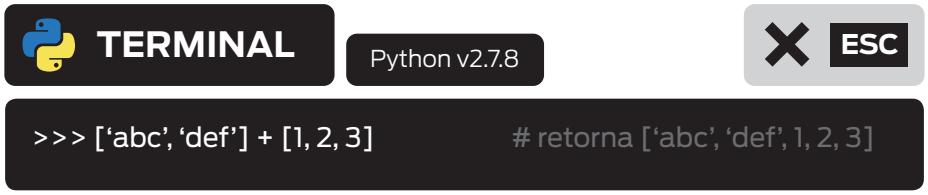
Listas são estruturas um pouco mais complexas. Elas podem guardar vários valores ao mesmo tempo, inclusive valores de diferentes tipos. São representadas usando colchetes e seus elementos são separados por vírgula. Exemplos de variáveis guardando listas seriam os seguintes:

TERMINAL Python v2.7.8 ESC

```
>>> lista_int = [1,2,3]
>>> lista_str = ['oi', 'ola', 'bom dia']
>>> notas = [6.2, 5.7, 8.3, 7.3]
>>> mistureba = [1, 2.0, '3', '4.0', 'cinco', [6]]
>>> lista_vazia = [ ]
```

Operações com listas:

Concatenação (+):



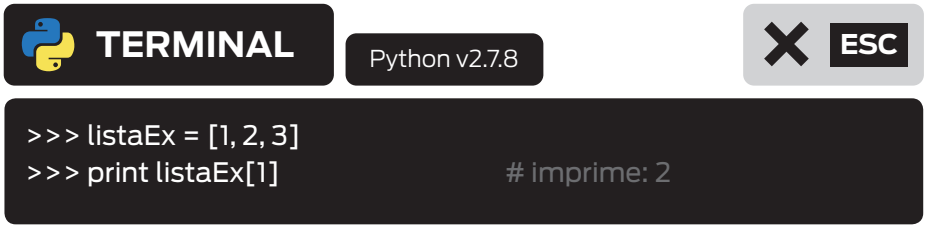
A partir de duas listas, geramos uma terceira contendo os elementos das listas envolvidas na operação.

Repetição (*):

A partir de uma lista e um inteiro, geramos outra lista contendo os elementos da lista original repetidos uma quantidade de vezes dada pelo inteiro que a acompanha na operação.

Indexação (lista[pos]):

De forma semelhante às strings, podemos obter elementos de uma lista a partir de uma determinada posição. Lembre-se que a primeira posição corresponde à posição 0, a segunda, à posição 1, e assim por diante.



Modificação de elemento (lista[pos] = novoElem):

Substitui o valor que estava na posição *pos* pelo valor de *novoElem*, modificando a lista.

Adição de novo elemento ao final da lista (lista.append(elem)):

Adiciona o elemento *elem* ao final de *lista*, modificando essa lista.

Adição de elemento em determinada posição da lista (lista.insert(pos, elem)):

Adiciona o elemento *elem* na posição *pos* em *lista*.

Deleção de elemento por posição (del lista[pos]):

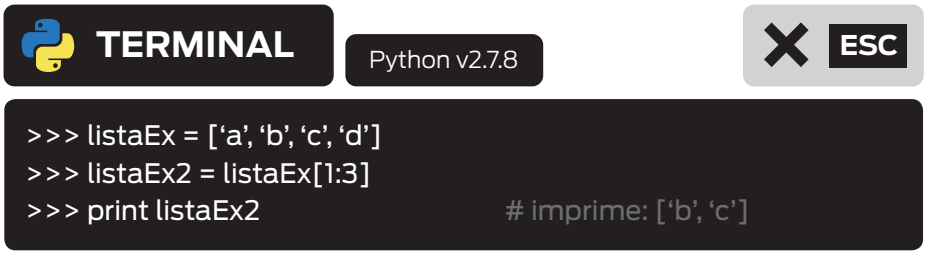
Elimina da lista o elemento que estiver na posição *pos*.

Remoção de determinado elemento (lista.remove(elem):

Elimina o elemento *elem* da lista. Se houver mais de uma ocorrência de *elem* na lista, remove a primeira (mais à esquerda).

Fatiamento (lista[pos1:pos2]):

Devolve uma nova lista com os elementos da lista original encontrados desde a posição *pos1* até a posição anterior a *pos2*.



NOTA: As propriedades colocadas na nota sobre fatiamento de strings também são validas para listas. Ex:

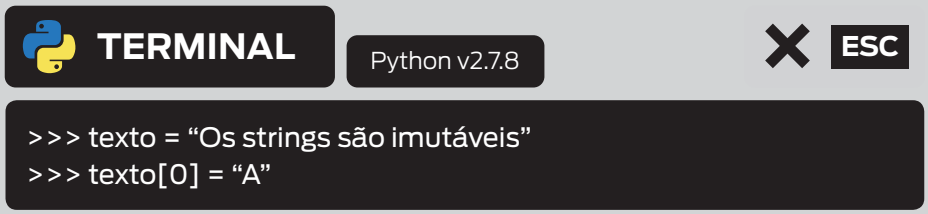
listaEx[pos:] criará uma nova lista com os elementos de *listaEx* encontrados desde a posição *pos* até o final.

>>>

Valores mutáveis e valores imutáveis

Listas são exemplos de valores mutáveis. Acabamos de ver que é possível adicionar, remover e modificar um elemento de uma lista qualquer.

Já as strings são imutáveis. Isto significa que, se tentarmos modificar uma string, aparecerá uma mensagem de erro. Por exemplo: Suponhamos que se tem a string "Os strings são imutáveis" guardada na variável *texto* e queremos modificá-la para "As strings são imutáveis", podemos ser tentados a pensar em mudar apenas o primeiro caractere de *texto* fazendo: texto[0] = "A". O código ficaria da seguinte forma:



Isto mostrará uma mensagem de erro após pressionarmos a tecla ENTER, pois strings, como foi mencionado, são imutáveis.

Embora as strings sejam imutáveis, é possível fazer algo do tipo:

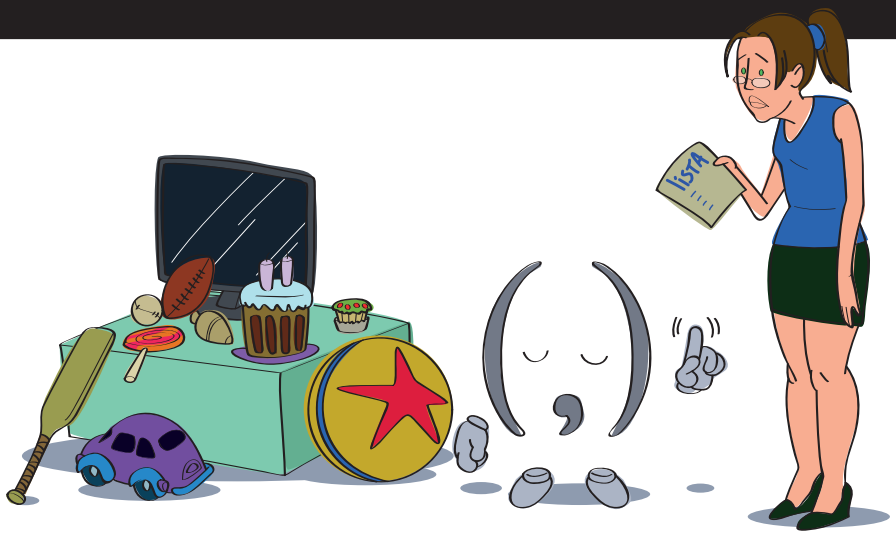
Isto não resultará em erro. Na verdade, não foi modificada a string original. Foi criada uma nova string e ela foi guardada na variável *texto*. Se quisermos imprimir essa variável, será impressa a string nova e não a velha. E o que acontece com a string velha? Se não estiver guardada em outra variável, será eliminada da memória do computador.

Tuplas

As tuplas são similares às listas, porém, não podem ser modificadas (são imutáveis), ou seja, não se podem colocar, trocar nem retirar elementos de uma tupla. Elas são representadas com parênteses e seus elementos devem ser separados por vírgulas. Exemplos de variáveis guardando tuplas seriam os seguintes

TERMINAL Python v2.7.8 ESC

```
>>> tupla1 = (‘eu’, ‘tu’, ‘ele’)
>>> tupla2 = (1, ‘1’, 1.0)
>>> tupla3 = (5,)                    # tupla com um elemento (note a vírgula)
>>> tupla_vazia = ( )                # tupla vazia
```



Operações com tuplas:

Concatenação (+):

TERMINAL Python v2.7.8 ESC

```
>>> (‘abc’, ‘def’) + (1, 2, 3)          # retorna (‘abc’, ‘def’, 1, 2, 3)
```

A partir de duas tuplas, geramos uma terceira contendo os elementos das duas tuplas originais.

Repetição (*):

TERMINAL Python v2.7.8 ESC

```
>>> (‘abc’, 2) * 3                      # retorna (‘abc’, 2, ‘abc’, 2, ‘abc’, 2)
```

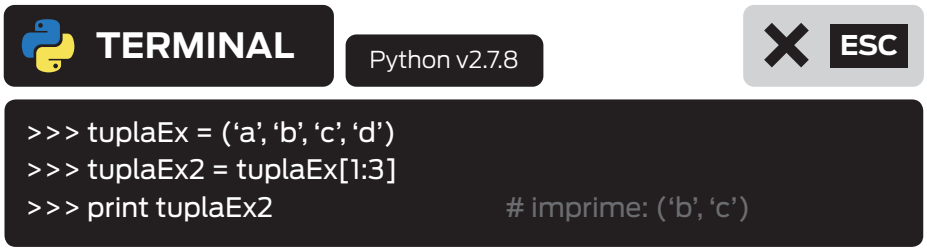
A partir de uma tupla e um inteiro, geramos outra tupla que contém os elementos da tupla original repetidos uma quantidade de vezes dada pelo inteiro que a acompanha na operação.

Indexação (tupla[pos]):

Assim como em strings e listas, a indexação também se aplica às tuplas. Podemos obter elementos de uma tupla a partir de uma determinada posição. Lembre-se que a primeira posição corresponde à posição 0, a segunda, à posição 1, e assim por diante. Ex:

Fatiamento (tupla[pos1:pos2]):

Aplicando esta operação a uma tupla, será devolvida uma outra tupla com os elementos da tupla original encontrados desde a posição *pos1* até a posição anterior à *pos2*. Ex:

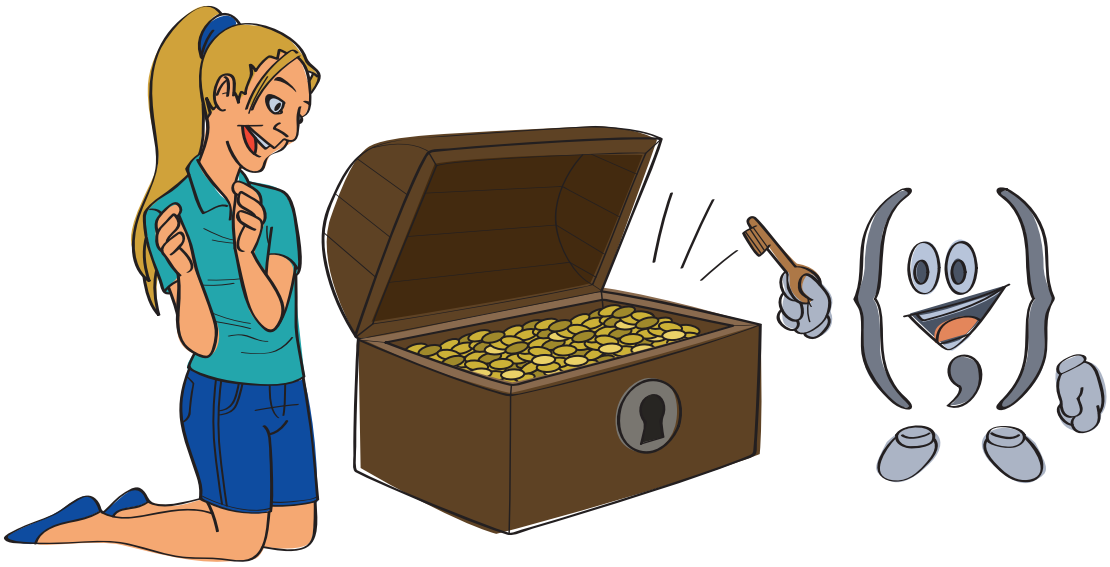


NOTA: O mesmo que foi falado sobre fatiamento para as strings e listas se aplica às tuplas.

Dicionários

Os dicionários são um pouco diferentes de tuplas e listas. Os dicionários são representados com chaves (**{}**). Eles são compostos por pares de dados (chave: valor). Cada par chave-valor deve ser separado dos demais por vírgulas. As chaves podem ser strings, inteiros,

floats e tuplas, enquanto que os valores podem ser de qualquer tipo. Perceba que o que chamamos "valor" aqui, não é exatamente o mesmo que vínhamos chamando anteriormente. Aqui "valor" é um valor (significado que vínhamos usando) que caracteriza uma chave. Com alguns exemplos isto ficará mais claro.



Exemplos de variáveis guardando dicionários seriam os seguintes:

TERMINAL Python v2.7.8 ESC

```
>>> dados = {'nome': 'Jose', 'sobrenome': 'Silva', 'idade': 22}
>>> dicionario_ingles = {'casa': 'house', 'livro': 'book', 'caneta': 'pen', 'xadrez': 'chess'}
>>> dicionario_vazio = {}                    # dicionário sem elementos
```

Operações com dicionários:

Indexação (dic[chave]):

A indexação nos dicionários é um pouco diferente se comparado com strings, tuplas e listas. Nestas, colocamos uma posição para obter o valor correspondente. Nos dicionários, em vez de uma posição, colocamos uma chave. Ex:

TERMINAL Python v2.7.8 ESC

```
>>> dicEx = {"v" : "vermelho" , "a" : "azul"}
>>> print dicEx["a"]                         # imprime: azul
```

Adição ou Modificação de chave (dic[chave] = valor):

Adiciona (se a chave não existir) ou modifica o valor associado à chave correspondente. Ex:


```
>>> dicEx = {'c' : 'Cruzeiro' , 'g' : 'Gremio' , 'a' : 'Atletico'}
>>> dicEx['c'] = 'Corinthians'
>>> dicEx['v'] = 'Vasco'
>>> print dictEx                    # imprime: {'a': 'Atletico', 'c': 'Corintians', 'g': 'Gremio', 'v': 'Vasco'}
```

Repare que, na hora de imprimir um dicionário, ele poderá parecer fora de ordem, mas o que interessa não são as posições e sim os pares chave-valor.

Deleção de chave(del dic[chave]):

Remove par chave-valor de *dic*.Ex:


```
>>> dicEx = {'um' : 1.0 , 'dois' : 2.0, 'tres' : 3.0}
>>> del dicEx['dois']
>>> print dicEx                              # imprime: {'um': 1.0, 'tres': 3.0}
```

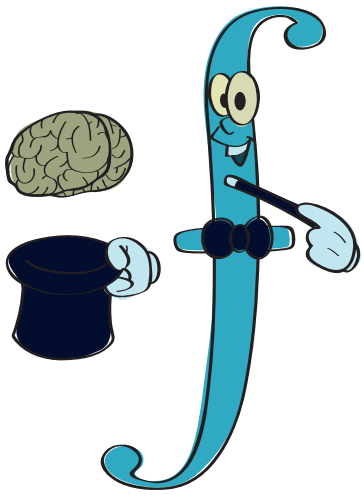
Vimos, desde os números até os dicionários, os principais tipos de valores com suas principais operações. Outros tipos de valores e/ou operações poderão ser apresentados nas atividades do curso. Fique atento!

Funções

Funções

Funções de Python

Funções... esse nome lhe parece familiar? Ah, sim! Da matemática! Veja este exemplo: $f(x) = x^2$. Podemos dizer que *f* é uma função que recebe um número e devolve o quadrado dele. Exemplos: f(1) = 1, f(2) = 4, f(3) = 9.



Na programação, podemos usar funções para agrupar um conjunto de instruções e realizar tarefas específicas, evitando ter que reescrever as instruções toda vez que se quiser realizar tal tarefa.

As funções podem receber alguns valores e trabalhar com eles. Abaixo temos a função *calcular_quadrado()**, que calcula o quadrado de um número, assim como a função matemática *f* comentada anteriormente.

Há várias novidades acontecendo por aqui que devemos analisar. Basicamente, definimos a função *calcular_quadrado()* e a executamos para obter o quadrado de 3. Quando trabalhamos com uma função, temos que pensar em duas coisas: defini-la e executá-la. Nas linhas 1, 2 e 3 estamos definindo a função. Na linha 5, ela é executada.

Definindo a função

A palavra-chave *def* é usada quando queremos definir uma função. Após, fornecemos um nome. Este nome deve seguir as mesmas regras que nomes de variáveis. Entre parênteses vão os parâmetros e, finalmente, dois pontos (:). Parâmetro é uma variável (x, neste caso) que recebe um valor quando a função é executada (veja a seção "Executando uma função").

O código que faz parte da função deve ir indentado (veja quadro indentação). Linhas de código com a mesma indentação pertencem a um mesmo bloco. Neste caso, pertencem ao bloco da função *calcular_quadrado()*. Isto ocorre nas linhas 2 e 3.

Dentro do bloco vemos um novo comando, o *return*. "Return", em inglês, significa retorne (devolva). Neste caso, retornará (quando a função for executada) o valor da variável *quadrado* que, por sua vez, guarda o valor do quadrado de *x*. Se este comando não existir dentro de uma função, a função retornará o valor *None*. "None" significa "nenhum" em inglês.

---

* *É comum colocar parênteses após o nome de uma função para explicitar que se trata de uma função e não de uma variável de outro tipo, como, por exemplo, um inteiro.*

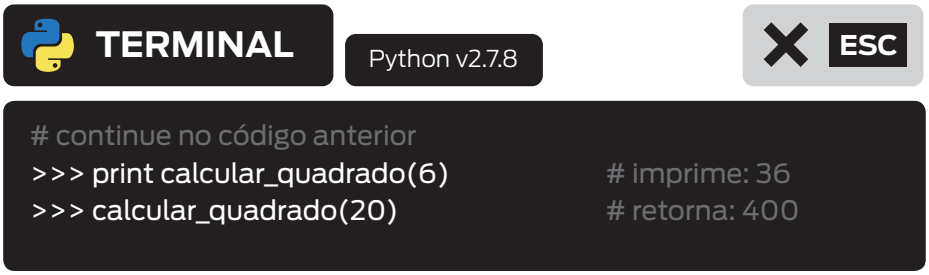
Executando a função

Na programação, é muito comum usar o termo "chamar" uma função para executá-la. Chamamos uma função colocando seu nome e, entre parênteses, os argumentos. Quando chamamos uma função, passa a ser executado o código que estiver no bloco da função. No exemplo, chamamos a função com o valor 3 (argumento), na linha 5. O parâmetro x na definição da função guardará esse valor e será, então, executado o bloco da função (linhas 2 e 3)

Com a executação do bloco, será retornado o valor 9. O valor retornado será guardado na variável *val* (linha 5) e impresso na tela (linha 6).

Uma função é definida uma vez e pode ser chamada quantas vezes desejarmos.

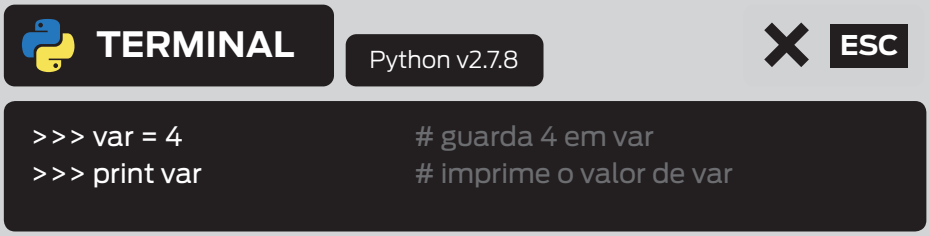
Continuemos com o código chamando esta função outras vezes.



>>>

Comando simples vs comando composto

Quando usamos o terminal interativo para criar nossos códigos (que é o que estamos fazendo até o momento), cada linha de código é executada após pressionarmos a tecla ENTER. Ex:



Estes são exemplos de comandos simples.

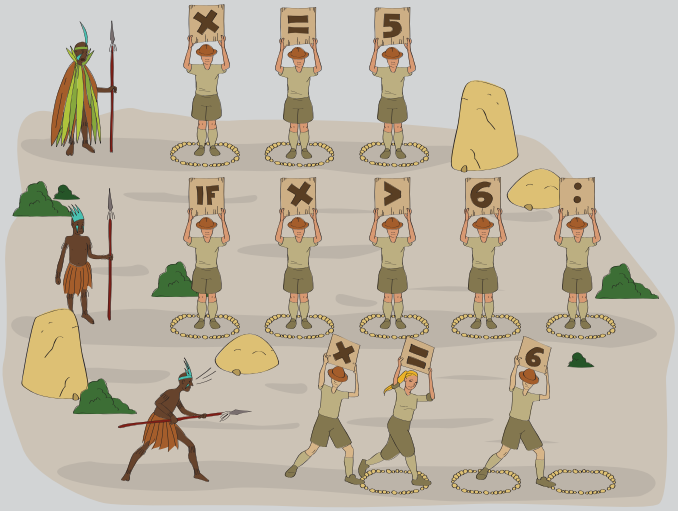
Um comando composto consiste em várias linhas de código, como, por exemplo, o comando def que define uma função. Os comandos compostos precisam de um ENTER extra para concluí-los. Por esta razão a linha em branco na linha 4 do último código. Com esse ENTER extra, Python percebeu que a definição da função terminava ali. No bloco do comando composto deverá existir pelo menos uma instrução.

Outra coisa a ser levada em conta nos comandos compostos é a indentação.

>>>

Indentação

É o espaçamento em relação à margem esquerda. As instruções dentro de comandos compostos devem ir indentadas para que Python saiba que pertencem a esse comando. Para indentar uma linha de código você pode usar tanto espaços como tabulações, mas não deve misturá-los. Usar espaços é mais aconselhável, pois diferentes ferramentas (editores, sistemas de email, impressoras...) tratam as tabulações de forma diferente.

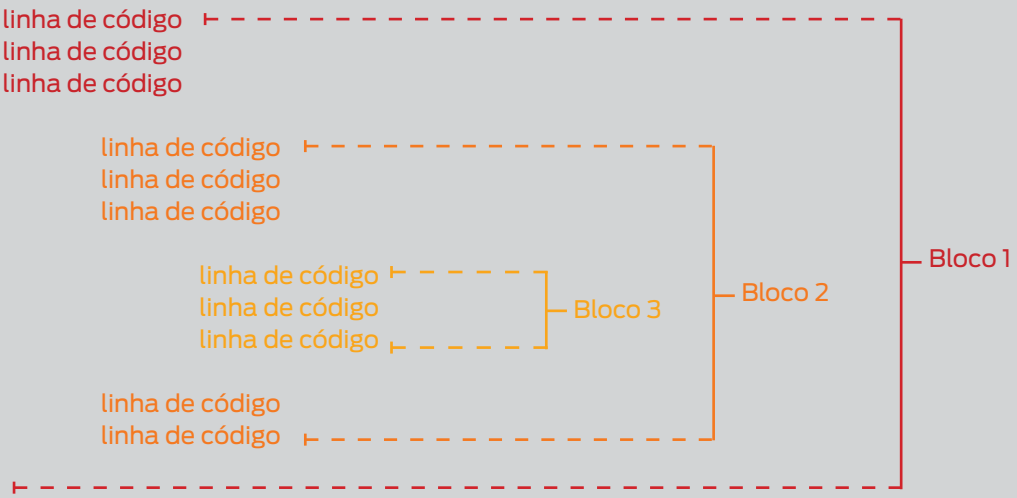


O bloco principal, aquele onde você começa um código, não poderá ser indentado, caso contrário teremos uma mensagem de erro. Ex:

TERMINAL Python v2.7.8 ESC

```
>>> print 'Tudo bem?'      # OK
>>>  print 'Tudo bem?'     # ERRADO. Existe um espaço antes do "print"
```

Dentro de um mesmo bloco a indentação deve ser a mesma. Porém, é possível ter blocos dentro de blocos, como será visto mais adiante.



A indentação não é usada apenas em funções, também a usamos em condicionais, loops e outras instruções compostas, como veremos ao longo do curso.

Variáveis Locais

Existem alguns comportamentos "sobrenaturais" quando trabalhamos com funções. Que tal um exemplo?

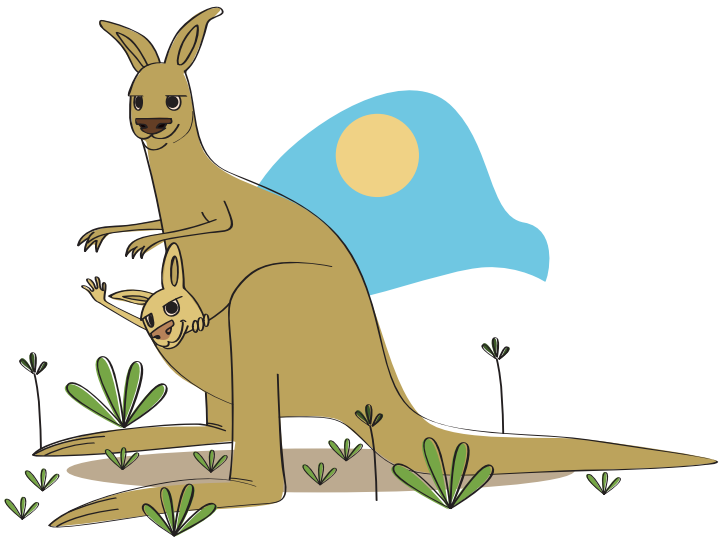
TERMINAL Python v2.7.8 ESC

```
>>> def funcaoTeste():
        val = 5

>>> funcaoTeste()
>>> print "val vale:", val
```

O que aconteceu aqui? Foi definida uma função que guarda o valor 5 na variável *val*, ela foi executada e impresso o valor de *val*. Mas por que a mensagem de erro parecendo dizer que *val* não está definida? O que é que deu errado?

O problema é que *val* é uma variável local. Parâmetros e variáveis criadas dentro de uma determinada função (usando "=", por exemplo) são chamadas variáveis locais e "vivem" apenas dentro da função. Quando a função termina de executar suas instruções, as variáveis locais "morrem". Como a instrução para imprimir está fora da função, apenas será enxergado o valor de *val* que estiver fora da função, se existir, senão aparecerá um erro.



Embora por fora de uma função não possamos enxergar uma variável local, o inverso é possível. Ou seja, de dentro de uma função podemos ver uma variável criada fora (chamada variável global) desde que ela tenha sido definida antes de chamar a função. Ex:

TERMINAL Python v2.7.8 ESC

```
>>> def funcaoTeste2():
        print “val vale:”, val

>>> val = 2
>>> funcaoTeste2()
```

É possível modificar o valor de uma variável global dentro de uma função, mas como este tipo de procedimento não é considerado uma boa prática de programação não o trataremos aqui. Caso seja necessário fazê-lo, será explicado em devida oportunidade.

Outras particularidades sobre funções, se surgirem, serão explicadas nas atividades do curso.

Funções de Python

Python possui algumas funções prontas (já definidas) que podem nos auxiliar em nosso trabalho. Vejamos algumas delas:

len(x)

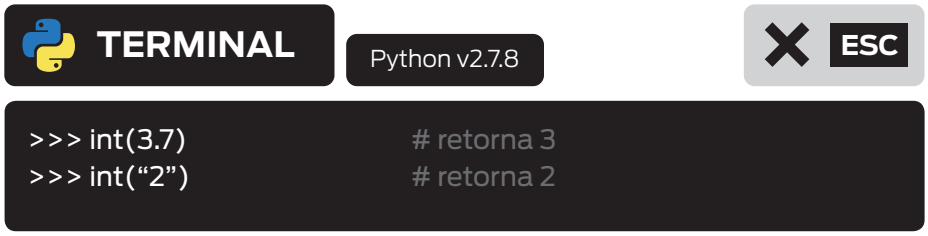
Esta função recebe uma string, lista, tupla ou dicionário e retorna o seu tamanho. Ex:

TERMINAL Python v2.7.8 ESC

```
>>> len(“casa”)          # retorna 4, pois esta string tem 4 caracteres
>>> len([‘casa’])        # retorna 1, pois esta lista possui um elemento
```

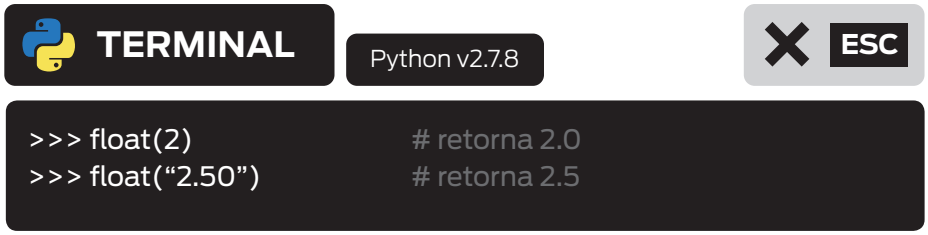
int(x)

Esta função recebe um número ou uma string (desde que contenha um inteiro) e retorna um inteiro. Ex:



float(x)

Esta função recebe um número ou uma string (desde que contenha um número) e retorna um float. Ex:

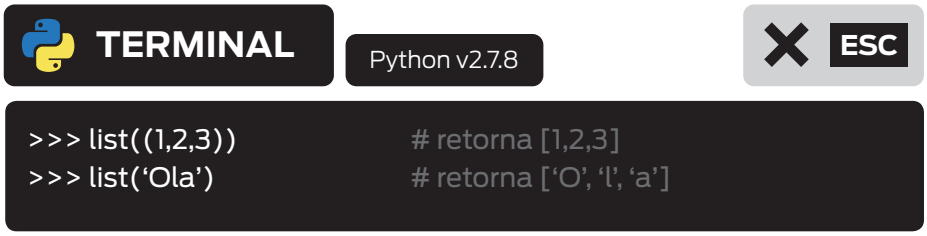


str(x)

Esta função converte o que recebe em uma string. Ex:

list(e)

Esta função converte o elemento que recebe em uma lista. Atenção, funciona apenas com elementos onde é possível fazer indexação (tuplas, strings, etc.) Ex:



tuple(e)

Esta função converte o elemento que recebe em uma tupla. Atenção, funciona apenas com elementos onde é possível fazer indexação (listas, strings, etc.) Ex:

round(x, y)

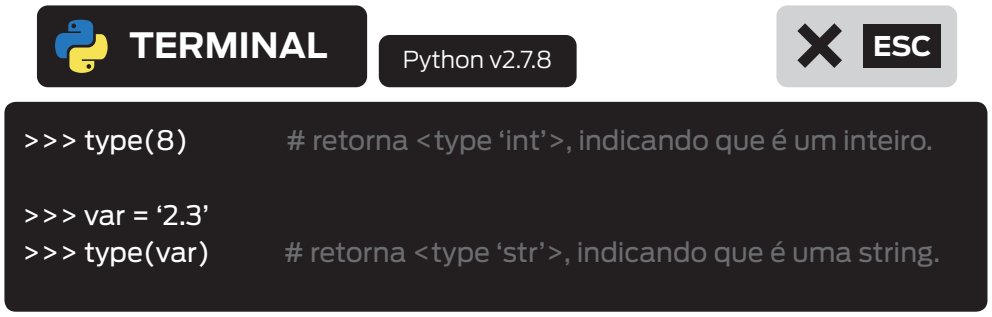
Esta função recebe um número (x) e, optativamente, um inteiro (y) e retorna o valor de *x* arredondado para *y* casas decimais. Ex:

range(a, b, c)

Esta função recebe um, dois ou até três inteiros e retorna uma lista que contém uma sequência que vai desde o número do primeiro argumento (incluindo-o) até o número do segundo argumento (excluindo-o) em passos dado pelo terceiro argumento. Ex:

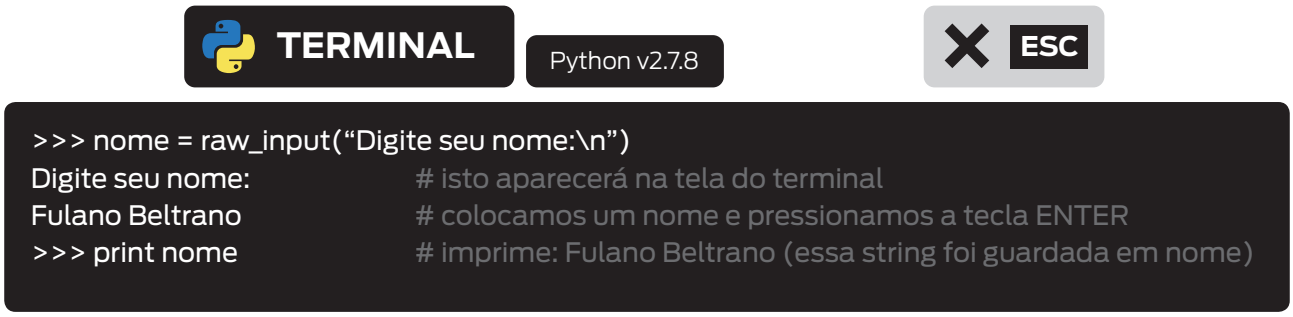
type(x)

Com esta função podemos determinar o tipo de algum valor (ou variável). Ex:

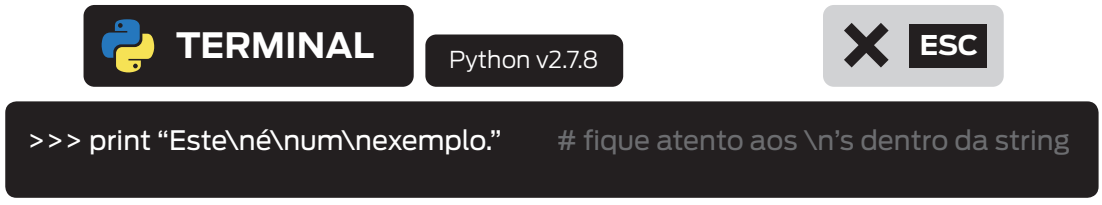


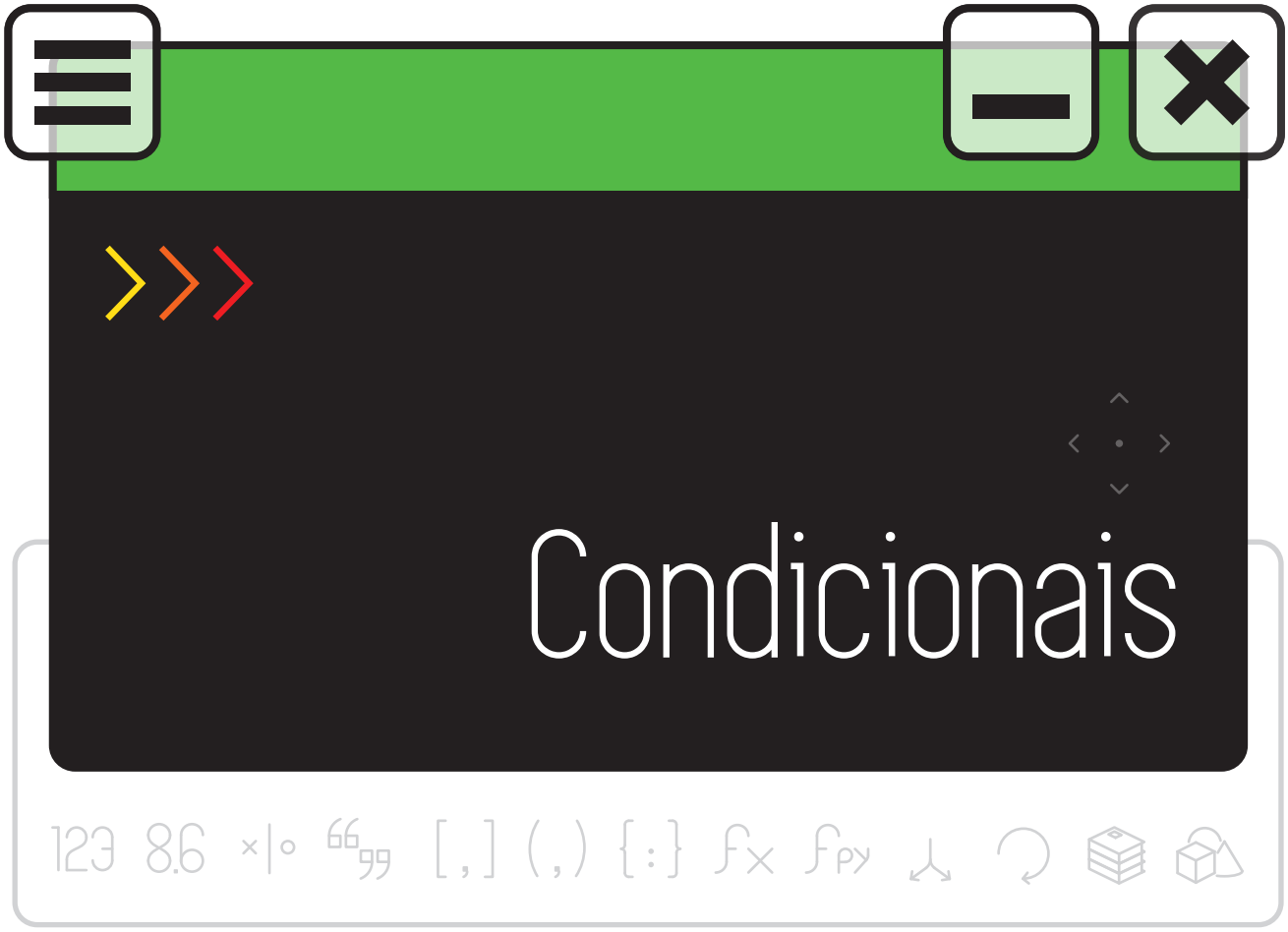
raw_input(mensagem)

Esta é uma função muito útil para receber dados por teclado. Ela tem um comportamento um pouco diferente das outras vistas. Ela mostrará a mensagem passada como argumento (se alguma), pausará a execução do programa para que o usuário coloque algum dado por teclado e pressione a tecla ENTER, transformará o que foi colocado em uma string e, finalmente, a retornará. Vejamos um exemplo:



NOTA: Neste exemplo usamos "\n". Esses dois caracteres dentro de uma string equivalem a um caractere "nova linha" em Python. Vejamos outro exemplo para esclarecer. Coloque a instrução abaixo e veja o que acontece quando é pressionado ENTER:

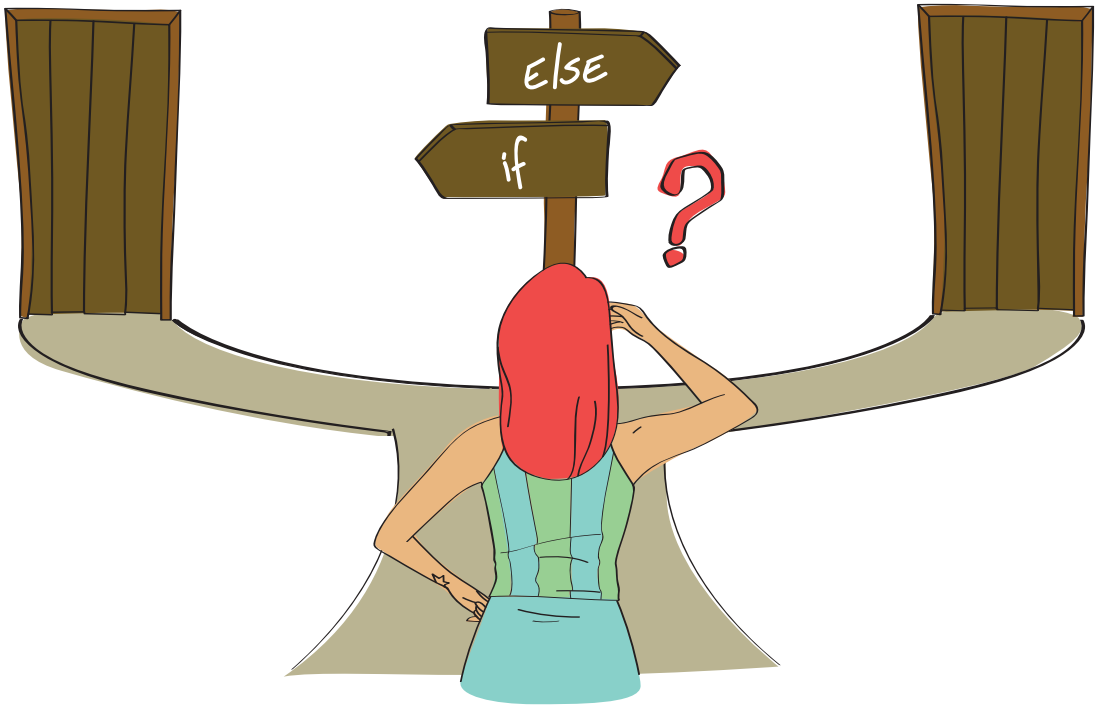




Condicionais

Até pouco tempo atrás, vimos que as linhas de código eram executadas sequencialmente. À medida que colocávamos uma instrução, esta era executada. Isto deixou de ser verdade quando usamos funções. Vimos que o código dentro de uma função seria executado apenas se a função fosse chamada. Se a função não fosse chamada, o código no seu interior não seria executado.

Algumas vezes podemos querer que uma instrução seja executada dependendo de certas condições. É neste tipo de situação que usamos condicionais.



Que tal fazer um programinha onde são comparados dois números? Serão passados dois números por teclado e será impresso o maior deles. Também vamos aproveitar e usar as funções *raw_input()* e *float()* de Python.

TERMINAL Python v2.7.8 **ESC**

```
>>> valor1str = raw_input('Entre com o primeiro valor: ')
Entre com o primeiro valor:          # isto aparecerá na tela, coloque um número e ENTER
>>> valor2str = raw_input('Entre com o segundo valor: ')
Entre com o segundo valor:           # entre com outro número e ENTER
>>> valor1 = float(valor1str)        # criará um float a partir da string
>>> valor2 = float(valor2str)        # criará um float a partir da string
>>> if valor1 > valor2:
        print 'o primeiro valor é maior do que o segundo'
elif valor1 < valor2:
        print 'o segundo valor é maior do que o primeiro'
else:
    print 'eles têm o mesmo valor'
# Aqui pressionamos ENTER para fechar a instrução do condicional.
```

O que foi impresso? Bom, aí depende dos números que você escolheu. Mas uma coisa é certa, foi impresso uma das possibilidades, não foi? Apesar de termos escrito várias instruções nem todas foram executadas. Vejamos como funciona o comando *if*.

Uma pequena aula de inglês vai ajudar a esclarecer. “If” significa “se”, “else” significa “senão” e “elif” significa... mmm... não tem nada no dicionário... Isto porque “elif”, na verdade, é uma abreviação de “else if”, que significa “senão se”. Então, traduzindo, a instrução ficaria:

- se *valor1* é maior do que *valor2*, imprima ‘o primeiro valor é maior do que o segundo’;
- senão, se *valor1* é menor do que *valor2*, imprima ‘o segundo valor é maior do que o primeiro’;
- senão, imprima ‘eles têm o mesmo valor’.

Quando usamos o comando *if*, devemos acompanhá-lo de uma expressão booleana (expressões que retornam valores booleanos), que será a condição a ser analisada. Se tal condição se cumpre, serão executadas as instruções desse bloco. Senão, será executado o bloco onde aparecer a primeira condição verdadeira dentro dos *elif's* seguintes. Se nenhuma condição for verdadeira, será executado o bloco *else*.

NOTA: Quanto tivermos a estrutura if...elif...else sempre será executado apenas um de seus blocos, aquele onde a condição for satisfeita ou o bloco *else*.

Laços (Loops)

Loops

Suponhamos que se queira imprimir cada letra da palavra “pneumoultramicroscopicossilicovulcanoconiótico”. Pelo conhecimento em programação que temos até agora, o código ficaria, mais ou menos, assim:

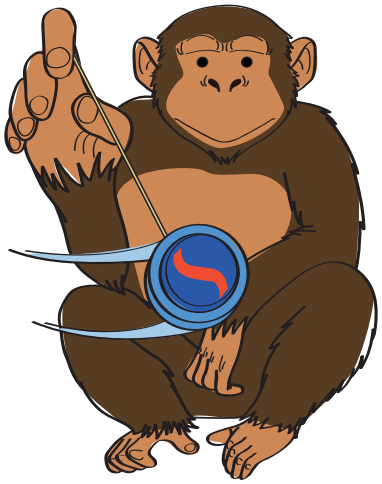

```
>>> print “p”
>>> print “n”
>>> print “e”
>>> print “u”
>>> print “m”
>>> print “o”                    # etc, etc... cansei... ainda falta muito?
```

Se você teve a paciência necessária para terminar, parabéns! Seu código fará o que é pedido. Missão cumprida! Porém, fica a sensação de que deveria existir uma maneira menos trabalhosa para fazer isso, não é mesmo? E existe!

TERMINAL Python v2.7.8 **ESC**

```
>>> for letra in “pneumoultramicroscopicossilicovulcanoconiótico”:
        print letra                          # aperte ENTER para terminar a instrução for
```

Se eu soubesse disso antes...!



Laços, também chamados de “loops”, servem para repetir a execução de um código, evitando ter que reescrevê-lo repetidas vezes. Eles permitem a execução de instruções repetidas vezes enquanto uma determinada condição for verdadeira ou para cada elemento de uma lista, tupla, dicionário ou string. Para o primeiro caso, usamos um loop *while* (“enquanto”, em inglês), para os outros, usamos um loop *for* (“para”, em inglês), que é o que foi usado em nosso exemplo.

Voltemos ao exemplo para explicá-lo. O comando *for* vem acompanhado de um nome (*letra*, no exemplo), a palavra-chave *in* e objetos tipo listas, tuplas, strings ou dicionários (ou funções que retornem objetos destes tipos, como é o caso da função de Python *range()*). A cada iteração (repetição), *letra* guardará um caractere da string “pneumoultramicroscopicossilicovulcanoconiótico”, na ordem, e, de acordo com a linha seguinte, esse valor será impresso. Ocorrerão tantas iterações quantos caracteres existam na string.

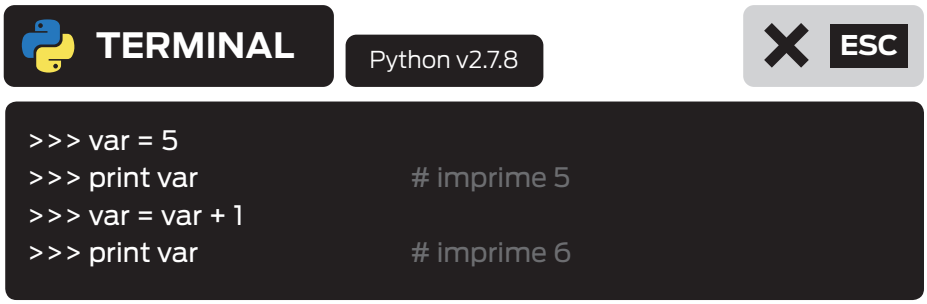
Outra forma de conseguir a execução repetida de um bloco é com o comando *while*. Ele funciona de forma um pouco diferente do que o comando *for*. Resolvamos o mesmo problema, mas desta vez usando o comando *while*. Usaremos a função *len()* para este exemplo.

TERMINAL Python v2.7.8 **ESC**

```
>>> pos = 0
>>> palavra = “pneumoultramicroscopicossilicovulcanoconiótico”
>>> while pos < len(palavra):
        print palavra[pos]
        pos = pos + 1                # aqui pressionamos ENTER para encerrar o bloco while
```

Podemos ver que o código faz exatamente o mesmo, porém foi escrito de forma diferente. Além de uma variável guardando a palavra de cujos caracteres queremos imprimir, também criamos a variável *pos*. Esta variável guardará a posição do caractere que queremos imprimir. A posição ira desde 0 até uma posição anterior ao comprimento de *palavra*, que corresponderá à posição do último caractere. Em seguida explicaremos melhor isto.

Outra coisa que podemos notar é a instrução **pos = pos + 1**. O que estamos dizendo aqui, basicamente, é: guarde na variável *pos* o valor que *pos* tiver nesse momento somado de 1. Inicialmente *pos* guarda o valor 0, após essa instrução, *pos* guardará o valor 1. Façamos um pequeno teste para esclarecer melhor a ideia.



Falemos, agora, sobre o comando *while*. Ele vem acompanhado por uma expressão booleana (pos < len(palavra), neste exemplo). Ali estamos comparando a posição nesse momento com o comprimento do valor guardado pela variável *palavra*, que é 20. O bloco *while* será executado sempre que *pos* for menor que 20, ou seja, até 19, que será a posição do último caractere. Mas o último caractere não deveria estar na posição 20? Não! Lembre-se que as posições começam no 0! Isto não acontece apenas com strings, também ocorre com tuplas e listas.

p	n	e	u	m	o	u	l	t	r	a	m	i	c	r	o	s	c	o	....	*caracteres*
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	....	*posições*

Dentro do bloco é impresso o caractere da posição nesse momento e, após, a posição é incrementada em uma unidade. Uma vez que a condição deixa de ser verdadeira (*pos* deixa de ser menor do que o comprimento da palavra), o bloco deixa de ser executado e passam a ser executadas, se houver, as instruções que o seguem.

NOTA: No exemplo, conseguimos imprimir cada letra perfeitamente, mas o que teria acontecido se nossa palavra tivesse acentos ou letras não comuns da língua inglesa?

Teste o seguinte:

TERMINAL Python v2.7.8 **ESC**

```
>>> len("coração")
```

Isto retornará 9 e não 7! Letras acentuadas (ou letras como "ç") podem ser consideradas como contendo mais de um caractere na versão 2 de Python. Fique atento!

>>>

For vs While

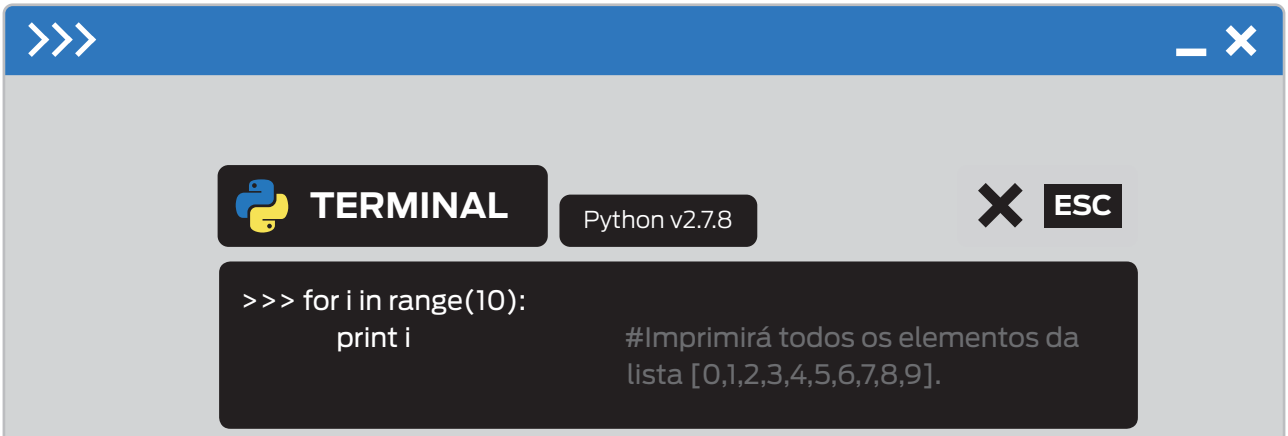
Vimos que podemos usar os loops *for* e *while* para trabalhar com comandos repetidos. Mas quando usar um ou outro?

Muitas vezes é uma escolha pessoal e podemos optar por qualquer um. Mas existem situações em que um deles leva vantagem.

Loops *for* são mais apropriados quando quisermos **repetir um código por um número determinado de vezes**. A seguir, dois exemplos.

TERMINAL Python v2.7.8 **ESC**

```
>>> for i in range(10):
        print 'i'                #Isto imprimirá 'i' 10 vezes.
```



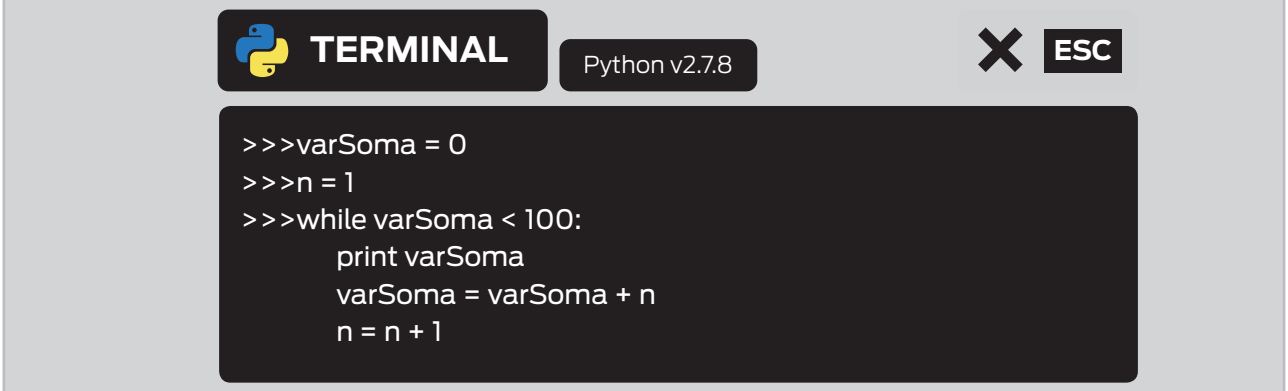
Note a diferença entre estes dois códigos. No primeiro imprimimos a string 'i' 10 vezes. No segundo imprimimos o valor que tem a variável *i* em cada iteração. Esta variável assumirá todos os valores da lista [0,1,2,3,4,5,6,7,8,9] nos dois exemplos, mas foi solicitado que estes valores sejam impressos apenas no segundo exemplo.

Já loops *while* são mais apropriados quando quisermos **repetir comandos um número indeterminado de vezes enquanto uma certa condição se satisfaça**.

Por exemplo:

Somar todos os números inteiros positivos enquanto o resultado da soma for menor que 100.

Poderíamos fazer o código da seguinte forma:



Aqui criamos a variável *varSoma*, que guardará a soma dos elementos analisados a cada iteração; e a variável *n*, que será o inteiro positivo a ser somado, também, a cada iteração..

Por que inicializamos o valor de *varSoma* com 0? Queremos que a variável *varSoma* guarde o valor da soma de alguns inteiros e não queremos que seu valor interfira nessa soma. Logo, o valor 0 é o ideal, pois qualquer número somado a 0 resulta no próprio número.

Se quiséssemos os produtos de inteiros positivos, um valor que não interferiria no resultado seria 1, pois qualquer número multiplicado por 1 resultaria no próprio número.

Por que inicializamos o valor de *n* com 1? Pois 1 é o primeiro inteiro positivo e sabemos que ele deve entrar na soma a ser realizada. Escolher os valores iniciais das variáveis (e quais usar) é essencial para que um loop *while* funcione corretamente. Isto somente é conseguido com prática ("A prática leva à perfeição").

Perceba que a condição que queremos para que o loop se repita é que o valor da variável *varSoma* seja menor que 100. Um detalhe importante sobre isto é que se o valor de *varSoma* nunca for maior ou igual a 100 o loop se repetirá indefinidamente (ou melhor, até você fechar a janela ou dar um ctrl-z ou ctrl-d, dependendo do sistema operacional, no terminal), ou, ainda, que apareça um erro. Por esta razão o valor dessa variável (*varSoma*) deve aumentar no interior do loop (para este exemplo).

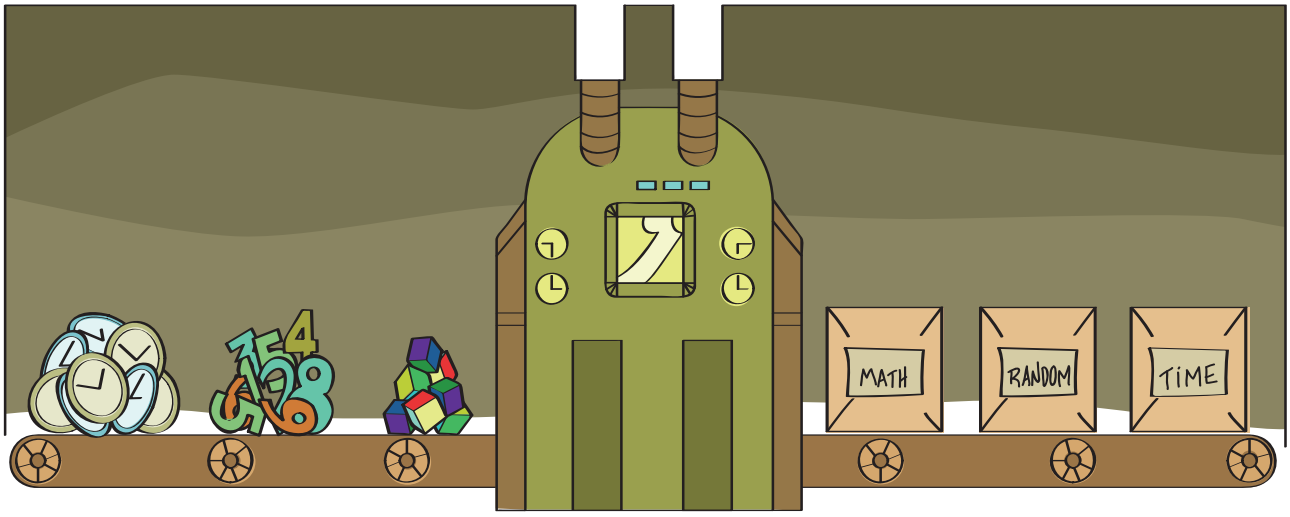
Dentro do loop somamos o valor de *varSoma* obtido até esse momento com o valor de *n*, também nesse momento, e incrementamos *n* para o seguinte inteiro para que seja somado na seguinte iteração. O comando *print* foi colocado para que você veja o valor de *varSoma* a cada iteração. Serão impressos os valores: 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78 e 91, pois o seguinte número a ser somado (o 14) resultaria em uma soma maior do que 100 (105, para ser mais preciso).

Módulos

Módulos

Módulos são arquivos que contêm códigos com funções e variáveis que podem ser aproveitados no código em que estamos trabalhando.

Usar códigos em módulos permite economizar trabalho, pois parte do que queremos fazer já está feito. Python contém vários módulos, como por exemplo, time, random, math, pygame etc. que podemos usar em nosso código. Também podemos criar nossos próprios módulos.



Para usar um módulo precisamos importá-lo. Usamos, para isto, a palavra-chave *import* seguido do nome do módulo. Importar módulos é uma das primeiras coisas que costumamos fazer em um código (quando queremos aproveitar código já existente). Para poder usar uma função ou variável do módulo que importamos, é preciso usar o nome do módulo, seguido de ponto (.), seguido do nome de uma função ou variável. Vejamos um exemplo:

TERMINAL Python v2.7.8 ESC

```
>>> import math              # é importado o módulo math
>>> print math.sqrt(16)      # imprime: 4.0
>>> print math.pi            # imprime: 3.14159265359
```

Aqui usamos a função *sqrt()* do módulo *math* para calcular a raiz quadrada de 16. Também imprimimos o valor da variável pi, também contida no mesmo módulo.

Existe outra forma de importar elementos de módulos. Usando a palavra-chave *from*, seguido do nome do módulo, de *import*, e das variáveis ou funções que nos interessam. O exemplo anterior ficaria, usando isto:

TERMINAL Python v2.7.8 ESC

```
>>> from math import sqrt, pi   # é importado sqrt e pi do módulo math
>>> print sqrt(16)              # imprime: 4.0
>>> print pi                    # imprime: 3.14159265359
```

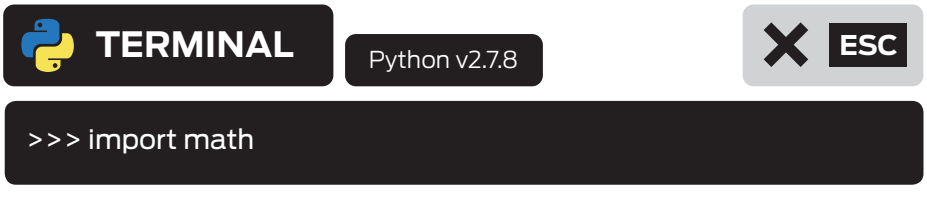
Aqui vemos que não precisamos do nome do módulo para usar uma função ou variável. A vantagem da outra forma é que permite saber, explicitamente, a que módulo pertencem os elementos. Se usarmos "*" depois de import, estaremos importando todas as funções e variáveis do módulo.

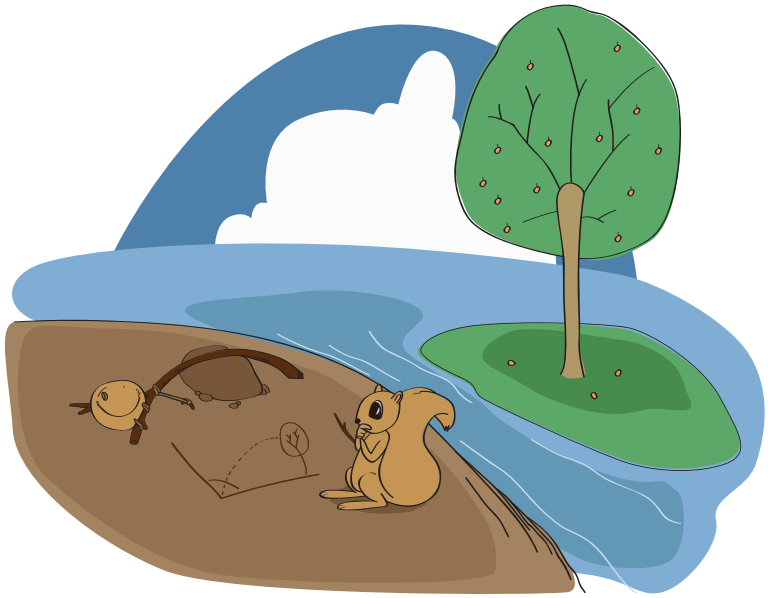
Podemos ter conhecimento das funções e variáveis contidas em um módulo usando a função *help()*.

Ali aparecerão um conjunto de funções (FUCTIONS) e variáveis (DATA) contidos no módulo. O problema é que as explicações estão em inglês. A seguir, explicaremos algumas funções e variáveis dos módulos que usaremos com mais frequência.

Módulo Math

Falamos, brevemente, sobre o módulo math na seção anterior. Desta vez, explicaremos algumas funções e variáveis, de este módulo, que usaremos com mais frequência no curso. Para realizar os exemplos que mostraremos, você deve importar, antes de tudo, o módulo math da seguinte forma:





Funções

sin(n): Corresponde à função seno da Matemática. Ela recebe um número como argumento. Este número deve ser o ângulo em radianos. Lembremos que 180º equivale a 3.1415...(pi) radianos. Ex:

NOTA: 1e-05 é 0.00001.

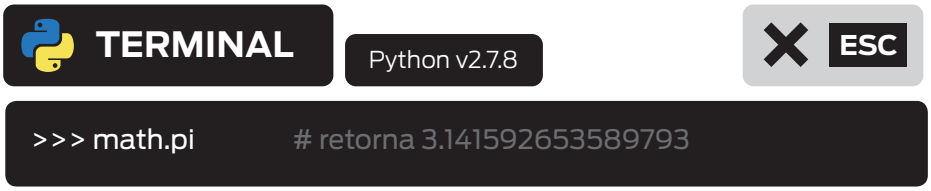
cos(n): Corresponde à função cosseno da Matemática. Semelhante à definição da função sin(). Ex:

tan(n): Corresponde à função tangente da Matemática. Semelhante às funções anteriores. Ex:

sqrt(n): Devolve a raiz quadrada do número passado como argumento. Ex:

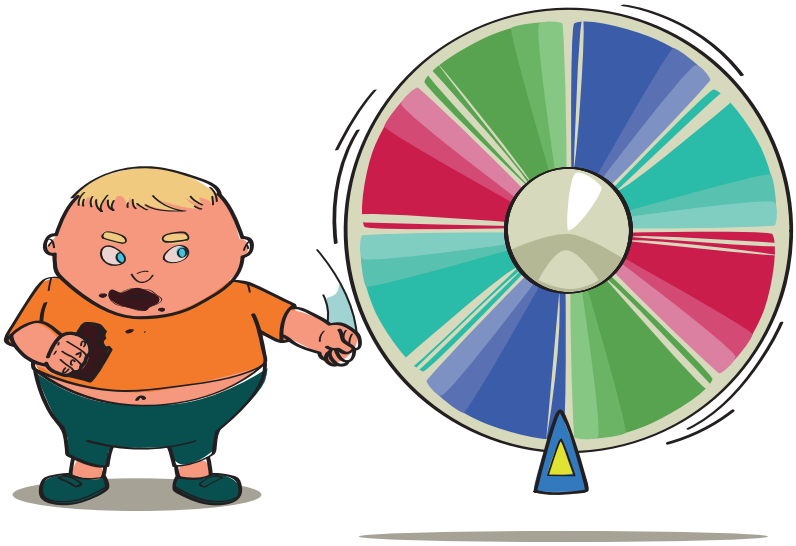
Variáveis

pi: Corresponde ao valor aproximado de p (pi) da matemática, que é o número resultante da relação do comprimento pelo diâmetro de uma circunferência. O valor desta variável é 3.141592653589793.



Módulo Random

Outro módulo bastante usado é o módulo random. Ele contém funções que tratam de processos aleatórios. Para realizar os exemplos, importe o módulo random.



Vejamos as principais funções:

choice(lista): retorna um item, escolhido aleatoriamente, da lista que recebe como argumento. Ex:

```
TERMINAL   Python v2.7.8                                   ESC

>>> lista = [1, 2, 3, 4, 5]
>>> random.choice(lista)      # retorna 4, por exemplo
```

randint(a, b): retorna um inteiro, escolhido aleatoriamente, entre os inteiros a e b (inclusive). Ex:

```
TERMINAL   Python v2.7.8                                   ESC

>>> random.randint(2,9)      # retorna 9, por exemplo
```

shuffle(lista): mistura os elementos da lista que recebe como argumento. Ex:

```
TERMINAL   Python v2.7.8                                   ESC

>>> listaEx = [1,2,3]
>>> random.shuffle(listaEx)
>>> print listaEx                  # imprime [2,1,3], por exemplo
```

Módulo Time

Neste módulo encontramos funções referentes ao tempo. Novamente, para realizar os exemplos você deverá importar o módulo *time*.

```
TERMINAL   Python v2.7.8                    ESC
>>> import time
```

Vejamos algumas funções:

time(): É uma das funções mais utilizadas. Ela retorna o tempo do sistema. Este tempo é o total de segundos transcorridos desde 1 de janeiro de 1970 até o instante em que a função é chamada. Ex:

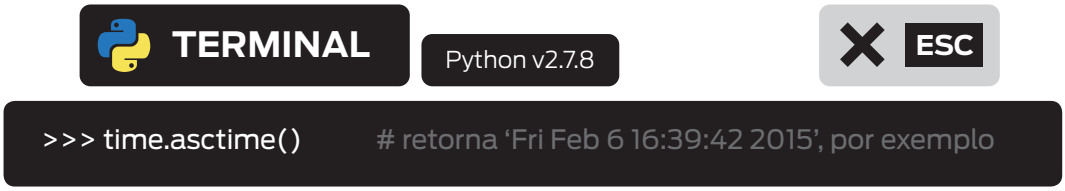
```
TERMINAL   Python v2.7.8                    ESC
>>> time.time()      # retorna 1423247588.297986, por exemplo
```

Este valor foi obtido no dia 06/02/2015 às, aproximadamente, 16:34:12. Quando você fizer o teste, este número será maior.

Esta função é muito útil para medir intervalos de tempo. Simplesmente, chama-se a função em dois instantes diferentes e poderá ser calculado o intervalo de tempo fazendo a diferença entre esses dois tempos.

sleep(t): provoca uma pausa de t segundos na execução do programa.

asctime(): mostra a data e hora locais dentro de uma string (em inglês).

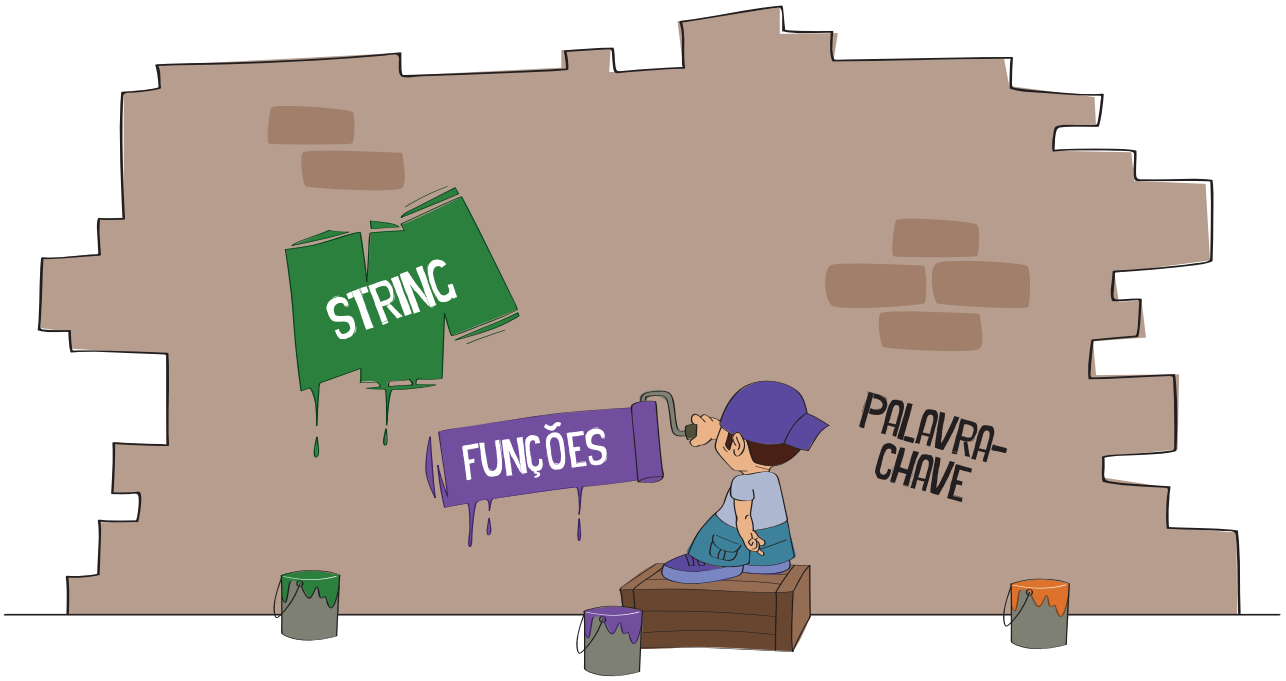


Como executar código por meio de arquivo fonte

Até agora vimos como executar um programa em modo interativo. Vimos que cada comando, seja simples ou composto, era executado à medida que era colocado no terminal. O modo interativo é ótimo para experimentar a linguagem e fazer pequenos códigos. Mas para grandes códigos se torna pouco prático, principalmente se quisermos guardá-los para executá-los outras vezes.

Agora vamos aprender como executar um programa usando o que chamamos de "arquivo fonte" ou "arquivo de código-fonte". Muito bem... Certamente você já ouviu falar em arquivos de texto, de áudio, de imagem, de vídeo... entre outros. Mas o que são arquivos fonte? Arquivos fonte, na programação, são arquivos que contêm código em alguma linguagem de programação.

Para criar um arquivo fonte precisamos de um editor de texto. O ideal seria usar um que contenha opções avançadas para a programação, como o Notepad++ ou o IDLE. Este último é instalado automaticamente quando instalamos Python em alguns sistemas operacionais (Windows, por exemplo) e podemos trabalhar nele tanto no modo interativo (Python Shell) como no editor de texto. O IDLE usa sintaxe colorida, colocando diferentes cores para as funções predefinidas, palavras-chave etc. Outra vantagem do IDLE é que coloca automaticamente a indentação, mas cabe ao usuário retirá-la (ou reduzi-la) quando necessário.



O uso do IDLE no modo interativo é muito parecido ao que vínhamos fazendo no terminal do jogo. Mas se abrirmos o editor de texto do IDLE (clicando em “File” e “New Window”) veremos que não aparecerá o símbolo “>>>”. Outra coisa, à medida que escrevermos código veremos que este não é executado instrução por instrução como no modo interativo. O código contido no arquivo só será executado quando clicarmos em “Run” e “Run Module”. Se o arquivo não foi salvo anteriormente, aparecerá uma janela pedindo para que seja salvo. Clicamos em “OK” e damos um nome para este arquivo. Perceba que será salvo com a extensão “.py”, indicando que é um arquivo com código Python. Se for usado outro editor, nós mesmos devemos colocar a extensão “.py”. Que tal criar um programinha no editor de texto do IDLE? Abra uma janela e mãos à obra!

A seguir temos dois códigos. Eles fazem exatamente a mesma tarefa. A diferença é que um está escrito para a versão 2 de Python e o outro para a versão 3. Dependendo da versão de Python que você tenha instalada, escreva o código correspondente no editor do IDLE, conforme explicado no parágrafo anterior, e execute-o.

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

import math

print "Este programa calcula raízes quadradas de números."
print "Para concluir sua execução pressione apenas Enter."
deve_continuar = True

while deve_continuar:
    valor = raw_input("Digite um número e pressione Enter: ")
    if valor == "":
        deve_continuar = False
    else:
        raiz = math.sqrt(float(valor))
        print raiz

print "\nFim da execução"
```

Código para Python 2.

```
import math

print ("Este programa calcula raízes quadradas de números.")
print ("Para concluir sua execução pressione apenas Enter.")
deve_continuar = True

while deve_continuar:
    valor = input("Digite um número e pressione Enter: ")
    if valor == "":
        deve_continuar = False
    else:
        raiz = math.sqrt(float(valor))
        print (raiz)

print ("\nFim da execução")
```

Código para Python 3.

NOTA: Outra forma de executar um arquivo com a extensão .py, sem o uso do IDLE, é abrindo um terminal (Linux ou Mac) ou o CMD do Windows, ir até o diretório onde o arquivo se encontra (ou usar o caminho absoluto) e, finalmente, escrever o seguinte comando no terminal:

python nome_do_arquivo.py (em Python 2 - ou se tiver apenas o Python 3 instalado)

ou

python3 nome_do_arquivo.py (em Python 3)

Também é possível, a partir de um terminal, executar o intérprete de Python, semelhante ao presente no jogo, colocando: python (ou python3).

Você consegue determinar o que este programa faz antes de tentar executá-lo? Execute-o para confirmar sua ideia. Se você conseguiu prever o que o programa faz, parabéns! Senão, não se preocupe, explicaremos abaixo o que é feito.

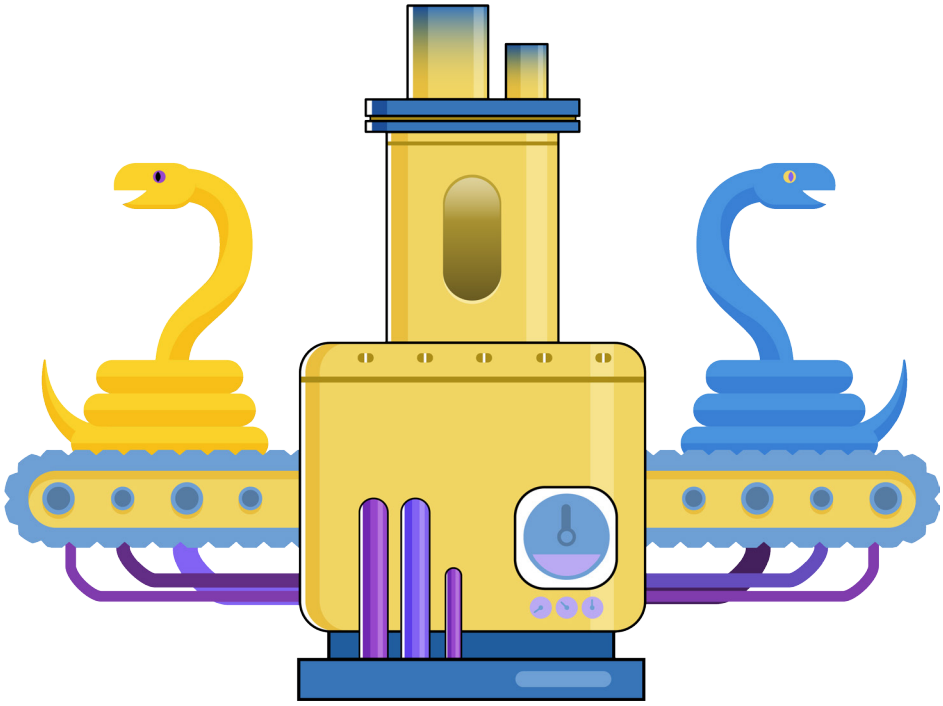
Basicamente, o programa calcula a raiz quadrada de números colocados por teclado. Primeiramente, importamos o módulo math, uma vez que queremos calcular raízes quadradas de números e lá existe uma função que faz exatamente isso. Após, definimos a variável *deve_continuar* que nos indicará se devemos continuar o loop ou terminá-lo. Inicialmente essa variável tem o valor booleano *True* para que possa entrar no bloco *while*, que será executado enquanto o valor de *deve_continuar* permaneça *True*. Dentro do loop, será obtido um número por teclado (se não for um número aparecerá uma mensagem de erro). Se nada for digitado, ou seja, se for pressionada apenas a tecla Enter, o valor de *deve_continua*r passará a ser *False*, concluindo o loop e imprimindo "Fim da execução". Senão, em *raiz* será guardada a raiz do número digitado e, logo a seguir, será impressa. Lembrando que o número digitado (guardado em *valor*) deve ser transformado em *float*, pois a função *raw_input()* retorna uma string e para calcular a raiz precisamos de um número.

Logo de cara podemos ver algumas diferenças entre um código no modo interativo e um em arquivo fonte:

- No modo interativo aparece o símbolo ">>>" para cada linha de instrução. Isto não acontece em arquivo de código-fonte.
- No modo interativo é necessário uma linha em branco para concluir uma instrução composta. As linhas em branco não são necessárias quando escrevemos em um arquivo de código-fonte, mas podem ser úteis para que o código fique mais claro.
- No modo interativo não é necessário usar o comando *print* para mostrar algum resultado na tela. Podemos, seguindo o exemplo, fazer: math.sqrt(9), onde será mostrado o valor 3.0. Em um arquivo de código-fonte devemos usar o comando *print* se quisermos que seja mostrado algum valor.
- Pode acontecer, na versão de Python que usamos no jogo, que ocorram erros devido ao uso de palavras com acentos. Se isto ocorrer, evite usá-los, assim como caracteres não presentes na língua inglesa ("ç", por exemplo).

Diferenças entre Python 2 e 3

No terminal do jogo usamos Python 2 por uma questão de compatibilidade. Mas usaremos a versão 3 de Python para trabalhar na segunda parte do curso (embora possa ser usada a versão 2, que já vem em muitos sistemas). Vejamos algumas diferenças entre estas versões.



- Em Python 2, o *print* é um comando. Já na versão 3 ele é uma função, por isso o uso de parênteses passa a ser obrigatório.

- Em Python 2, quando dividimos dois inteiros entre si usando "/" obtemos um inteiro. Em Python 3, esta divisão resulta em um *float*.

- Em Python 2 é usada a função *raw_input()* para criar uma string a partir da entrada de dado por teclado. Em python 3 a função equivalente é a *input()*.

> **Atenção:** Python 2 também possui a função *input()*, mas com um comportamento diferente. Ela não retorna uma string e sim o tipo que for inserido por teclado. Ex: Na versão 2, se for inserido 5 (sem aspas), *input()* retornará o inteiro 5. Enquanto que na versão 3 (ou *raw_input()*) retornará a string '5'.

- Em Python 2, a função *range()* retorna uma lista. Em Python 3 a função existe, mas não retorna uma lista. Porém, podemos criar uma lista a partir dela como mostrado no seguinte exemplo. Ex: criar uma lista com os inteiros de 0 a 9:

Python 2: range(10)

Python 3: list(range(10))

Para o loop *for* o uso é o mesmo nas duas versões. Ex:

for i in range(10): ...

- Em Python 2 é necessário uma linha de código especial no começo do arquivo para poder aceitar acentos (# -*- coding: UTF-8 -*-). Em Python 3, os acentos e caracteres especiais, como “ç”, podem ser usados sem problemas.

Créditos:

Coordenação: Gilson Oliveira Barreto
Conteúdo: Rodrigo Castro
Projeto gráfico: Igor Avelar
Editoração eletrônica: Fernando Simon
Ilustração: Biagy
Revisão ortográfica: Maria Lourença