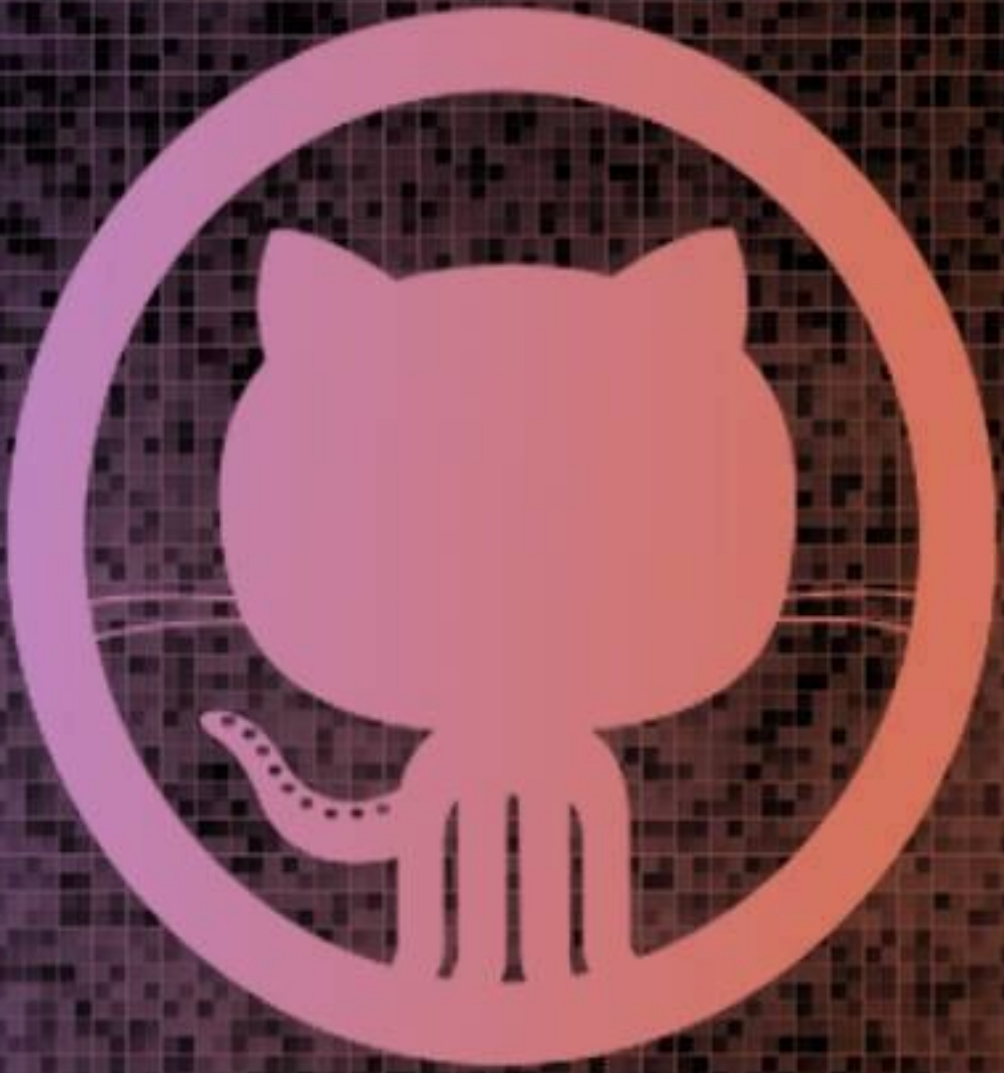
ESCOLA DE VERÃO 2015

26/01 a 30/01



DESENVOLVIMENTO COLABORATIVO

Patrocínio

novatec

PET Ciência da Computação

Pró-Reitoria de Graduação

Universidade Federal do Rio Grande do Norte

Table of Contents

Curso de Verão

Python

Este material foi criado com base na documentação oficial e no Livro Introdução à Programação com Python, de Nilo Ney Coutinho Menezes publicado pela Novatec (ótimas fontes de estudos).

Ele servirá como fonte de estudo no curso de verão 2015 realizado pelo PET CC.

Esperamos que seja util para seus estudos e como fonte de consulta. Os seguintes pontos serão vistos durante o curso:

Abrindo o Apetite

FONTE: PyDoc

Se você trabalha muito com computadores, acabará encontrando alguma tarefa que gostaria de automatizar. Por exemplo, você pode querer fazer busca-e-troca em um grande número de arquivos de texto, ou renomear e reorganizar um monte de arquivos de fotos de uma maneira complicada. Talvez você gostaria de escrever um pequeno banco de dados personalizado, ou um aplicativo GUI especializado, ou um jogo simples.

Se você é um desenvolvedor de software profissional, pode ter que trabalhar com várias bibliotecas C/C++/Java, mas o tradicional ciclo escrever/compilar/testar/recompilar é muito lento. Talvez você esteja escrevendo um conjunto de testes para uma biblioteca e está achando tedioso codificar os testes. Ou talvez você tenha escrito um programa que poderia utilizar uma linguagem de extensão, e você não quer conceber e implementar toda uma nova linguagem para sua aplicação.

Python é a linguagem para você.

Você poderia escrever um script para o shell do Unix ou arquivos em lote do Windows para algumas dessas tarefas, mas scripts shell são bons para mover arquivos e alterar textos, mas não adequados para aplicações GUI ou jogos. Você poderia escrever um programa em C/C++/Java, mas pode tomar tempo de desenvolvimento para chegar até um primeiro rascunho. Python é mais simples, está disponível em Windows, Mac OS X, e sistemas operacionais Unix, e vai ajudá-lo a fazer o trabalho mais rapidamente.

Python é fácil de usar, sem deixar de ser uma linguagem de programação de verdade, oferecendo muito mais estruturação e suporte para programas extensos do que shell scripts oferecem. Por outro lado, Python também oferece melhor verificação de erros do que C, e por ser uma linguagem de muito alto nível, ela possui tipos nativos de alto nível: dicionários e vetores (arrays) flexíveis. Devido ao suporte nativo a uma variedade de tipos de dados, Python é aplicável a um domínio de problemas muito mais vasto do que Awk ou até mesmo Perl, ainda assim muitas tarefas são pelo menos tão fáceis em Python quanto nessas linguagens.

Python permite que você organize seu programa em módulos que podem ser reutilizados em outros programas escritos em Python. A linguagem provê uma vasta coleção de módulos que podem ser utilizados como base para sua aplicação — ou como exemplos para estudo e aprofundamento. Alguns desses módulos implementam manipulação de arquivos, chamadas do sistema, sockets, e até mesmo acesso a bibliotecas de construção de interfaces gráficas, como Tk.

Python é uma linguagem interpretada, por isso você pode economizar um tempo considerável durante o desenvolvimento, uma vez que não há necessidade de compilação e vinculação (linking). O interpretador pode ser usado interativamente, o que torna fácil experimentar diversas características da linguagem, escrever programas “descartáveis”, ou testar funções em um desenvolvimento bottom-up. É também uma útil calculadora de mesa.

Python permite a escrita de programas compactos e legíveis. Programas escritos em Python são tipicamente mais curtos do que seus equivalentes em C, C++ ou Java, por diversas razões:

- os tipos de alto nível permitem que você expresse operações complexas em um único comando;
- a definição de bloco é feita por indentação ao invés de marcadores de início e fim de bloco;
- não há necessidade de declaração de variáveis ou parâmetros formais;

Python é extensível: se você sabe como programar em C, é fácil adicionar funções ou módulos diretamente no interpretador, seja para desempenhar operações críticas em máxima velocidade, ou para vincular programas Python a bibliotecas que só estejam disponíveis em formato binário (como uma biblioteca gráfica de terceiros). Uma vez que você tenha sido fisgado, você pode vincular o interpretador Python a uma aplicação escrita em C e utilizá-la como linguagem de comandos ou extensão para esta aplicação.

A propósito, a linguagem foi batizada a partir do famoso show da BBC “Monty Python’s Flying Circus” e não tem nada a ver com répteis. Fazer referências a citações do show na documentação não é só permitido, como também é encorajado!

Agora que você está entusiasmado com Python, vai querer conhecê-la com mais detalhes. Partindo do princípio que a melhor maneira de aprender uma linguagem é usando-a, você está agora convidado a fazê-lo com este tutorial.

Uma Introdução ao Mundo Python

O que é Python?

Segundo seu criador, Guido van Rossum:

> "Python é uma linguagem de programação poderosa e fácil de aprender. Ela possui estruturas de dados de alto nível e uma simples mas eficiente abordagem da programação orientada a objetos. Sua elegante sintaxe e tipagem dinâmica juntamente com seu interpretador nativo fazem dela a linguagem ideal para scripting e desenvolvimento rápido de aplicações em diversas áreas sob várias plataformas."

Bem, achamos está uma ótima definição para o Python, principalmente por ser dado pelo seu próprio criador.

Por que Python?

Até agora você deve ter percebido várias características desta linguagem que a tornam uma ótima opção. Ela é simples, clara, objetiva. O que não a torna menos poderosa que outras linguagem.

Python também é um software livre, ou seja, está disponível para ser utilizada gratuitamente (agradeça ao Python Foundation e aos seus vários colaboradores). Além disso, ela não te limita a um sistema operacional ou a uma arquiterura... mais liberdade.

Se você ainda está se perguntando quem usa Python ou para que vou usar essa linguagem... bem, inteligência artificial, banco de dados, biotecnologia, animação 3D, aplicativos móveis, jogos, aplicativos web, estas são algumas das utilidades do Python. Além disso, grandes empresas e aplicativos utilizam a linguagem, como a Globo.com, Netflix, Google, Amazon Web Service, etc.

Mas adiante veremos outros motivos para considerar Python uma ótima linguagem de programação tanto para iniciantes quanto para programadores experiêntes.

Tipagem Dinâmica

Para entender tipagem dinâmica, precisamos ter em mente um dos conceitos básicos de programação, o conceito de **variável**. A variável é uma entidade que possui um conjunto de atributos como nome, tipo e valor. Por exemplo se imaginarmos a memória de um computador como sendo uma estate de um depósito, em que cada compartimento representa um espaço de memória, podemos também imaginar uma váriável como sendo uma caixa que pode ser armazenada nessa estante.

Nessa analogia, um pouco surpeficial, dizemos que o nome da váriável é um rótulo que está associado a ela e quando formos procurar-la temos que buscar por esse rótuno. O tipo é basicamente a especificação de quais informações podem ser guardadas na caixa, ou em outras palavras, o tipo e tamanho do material que pode ser armazenado nela. Já o valor é o conteúdo armazenado dentro da caixa. Então, se dermos o nome `idade` a uma váriável e quisermos atribuir um valor a ela devemos fazer `idade = 19`. Dessa forma estamos dizendo que a variável `idade` está recebendo o valor `19` que por sua vez é do tipo inteiro (`int`).

Agora que o conceito de variável está mais claro, quando estamos programando em Python podemos ter variáveis armazenando valores de diversos tipos, desde valóres numéricos, lógicos, até mesmo caracteres e textos (strings). Alguns desses tipos serão vistos nas próximas seções por meio de exemplos.

Na linguagem Python, não há a necessidade de declarar o tipo da variável; para criar uma, basta atribuir um valor a ela. Como em `idade`. Nesse caso, o tipo da variável pode variar durante a execução do programa.

IMPORTANTE : Como foi dito: a variável possui um tipo. Ao declarar `idade = 19`, a variável `idade` assume o tipo `int`. A questão aqui, é que o tipo que a variável assume pode variar ao logo da execução, em função do dado que ela armazena.

Linguagem Interpretada X Linguagem Compilada

Esse assunto para muitos desenvolvedores ainda é algo um pouco confuso e rende muitas discussões entre várias comunidades de programadores. É importante falarmos sobre ele para que possamos fazer uma comparação do desempenho do Python e de outras linguagens, e o processo de interpretação e compilação irá nos ajudar nisso.

No processo de Compilação, programas podem ser traduzidos para linguagem de máquina, que pode ser executada diretamente no computador. Basicamente, nós teremos o que chamamos de código fonte, esse código será analizado lexicalmente e sintaticamente na tentativa de encontrar possíveis erros, esse processo gera um código intermediário (não mais na linguagem do código fonte). Nesse momento, mais uma análise é feita no código intermediário, a análise semântica. Por fim, o código intermediário passa por uma otimização se transformando na linguagem de máquina. A vantagem deste método é que a execução se torna bastante rápida, já que o processo de tradução já está completo.

Já no processo de Interpretação, temos uma abordagem diferente. Nesse caso o programa é interpretado por outro programa chamado interprete. O programa interprete vai agir como uma simulação de uma máquina cujo o ciclo de execução trata com linguagem de alto nível em vez de instruções de máquina. Essa simulaçao fornece uma máquina virtual para o linguagem. Uma das vantagens desse método é a detecção de erros, isso por que se houver alguma inregularidade uma mensagem de erro será exebida em tempo de execução, e irá se referir aquela unidade específica do código. Porém, uma desvatagem é o tempo gasto no processo se formos compará-lo ao processo da compilação. Isso por que toda vez que vamos executar nosso programa, ele passará por esse mesmo processo de interpretação.

Bloco por Indentação

Se você tem alguma experiência com Python, então deve saber que diferentes de outras linguagens que utilizam marcadores para delimitar trechos de códigos, por exemplo:

```
if (true) {
    /* código do bloco */
}
```

Nele basta você usar a indentação para demarcar os blocos. Como no exemplo:

```
if (true):
    # bloco de código
    # (ao fim da seção indentada
    # termina o bloco)
# próxima instrução (após o if)
```

É importante perceber que isso vai "exigir" certos cuidados. Para alguns, utilizar a identação (e deixar de lado as comuns instruções delimitadoras) pode parecer estranho, e de certa forma desorganizado. Mas com o tempo, você vai notar que o código vai ficar ainda mais legível e você acabará tendo que ser ainda mais organizado para não gerar erros.

Importante [1] : É importante convencionar se a indentação será feita por uma tabulação ou por um número determinado de espaços, já que todas as pessoas editando um programa Python devem usar o mesmo tipo. Uma boa dica que evita confusão é usar 4 espaços para cada nível.

Variáveis e Entradas de Dados

Até esse ponto, você já entendeu o que é uma variável. Agora, precisamos trabalhar alguns conceitos mais amplos, como regras para nomear variáveis, ou as características e recursos que as variáveis assumem ao serem tipadas.

Nomes de variáveis

No caso de Python, nomes de variáveis podem conter números, porém é obrigatório que você comece com letra ou com o símbolo `_`.

Na versão 3 do Python, é permitida a utilização de acentos em nomes de variáveis (isso por que os programas são interpretados utilizando o conjunto de caracteres UTF-8).

Outro ponto importante que devemos chamar atenção é para as palavras reservadas. Palavras reservadas definem as regras e a estrutura da linguagem e não podem ser usadas como nomes de variáveis [2].

Python tem 29 palavras reservadas:

and	**def**	**exec**	**if**	**not**	**return**
assert	del	finally	import	or	try
break	elif	for	in	pass	while
class	else	from	is	print	yield
continue	except	global	lambda	raise	

Você também pode obter a lista de palavras reservadas com estes comandos: `import keyword; print keyword.kwlist`.

Variáveis Numéricas

Quando uma variável armazena um número inteiro ou ponto flutuante ela pode ser chamada de variável numérica.

```
>>> largura = 20
>>> algura = 5*9
>>> largura * altura
900
```

Em operação com operandos de diferentes tipos, há a conversão do inteiro para ponto flutuante:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Números complexos também são suportados; números imaginários são escritos com o sufixo j ou J. Números complexos com parte real não nula são escritos como (real+imagJ), ou podem ser criados pela chamada de função complex(real, imag).

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Algumas funções realizam a conversão para float e inteiro, float(), int() e long(), porém no caso dos números complexos elas não funcionam. Nesse caso use abs(z) para obter sua magnitude (como um float) ou z.real para obter sua parte real.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
```

No modo interativo (basta abrir o terminal e digitar `python`) o valor da última expressão exibida é atribuída a variável `_`. Assim, ao utilizar Python como uma calculadora, fica mais fácil prosseguir com os cálculos, por exemplo:

```
>>> taxa = 12.5 / 100
>>> preco = 100.50
>>> preco * taxa
12.5625
>>> preco + _
113.0625
>>> round(_, 2)
113.06
```

Essa variável especial deve ser tratada como somente para leitura pelo usuário. Nunca lhe atribua explicitamente um valor

— do contrário, estaria criando uma outra variável (homônima) independente, que mascararia a variável especial com seu comportamento mágico [3].

Variáveis lógicas

Existem situação em que você precisa de uma informação simple sim ou não, falso ou verdadeiro, correto ou incorreto, etc. Nesses casos existe um tipo de variável chamado booleano (ou lógico). Em python, escrevemos `True` para verdadeiro e `False` para falso **(importante perceber que a primeira letra, T e F, são escretas em maiúsculo)**.

```
resultado = True
aprovado = False
```

Operadores

Nós também temos disponível um conjunto de operadores relacionais para realizarmos comparações lógicas.

Operador	Operação
==	igualdade
>	maior que
<	menor que
!	diferente
>=	maior ou igual
<=	menor ou igual

O resultado de uma comparação é um valor do tipo lógico, ou seja, `True` ou `False`[3].

```
>>> var1 = 1        # atribuimos 1 a var1
>>> var2 = 5        # atribuimos 5 a var2
>>> var1 == var2    # var1 é igual a var2?
False
>>> var1 < var2     # var1 é menor que var2?
True
>>> var1 != var2    # var1 é diferente de var2?
True
>>> var1 > var2     # var1 é maior que var2?
False
```

Note que usamos o símbolo `#` para representar comentários, o que estiver depois deste símbolo na linha será ignorado pelo interpretador Python. Já que tocamos no assunto "comentários em código", é importante chamar atenção para o seu uso durante o desenvolvimento. Tenha e mantanha o costuma de comentar trechos do seu código, não precisa colocar um comentário a cada linha (a não ser que você queira), mas é interessante deixar pequenas explicações sobre partes do código para quando você for lê-lo daqui a 1 semana, 1 mês, anos ou para quando alguém for estudá-lo. Também é indicável que você identifique os programas com seu nome, data em que começou a escrevê-lo e em que fez a última modificação, além de possíveis fontes e referências que usou para o desenvolvimento.

Também é possível guardar em variáveis o resultado de expressões e comparações lógicas:

```
>>> nota = 9
>>> media = 6
>>> aprovado = nota > media
True
```

Quiz

Question 1 of 1

Agora, só para conferir se você entendeu bem, marque somente as expressões incorretas: Tomando a = 4, b = 10, c = 5, d = 0 e f = 5.0

- [ ] a == b True
- [ ] a < b True
- [ ] a < b False
- [ ] c >= d True
- [ ] c == f False
- [ ] a < d False
- [ ] c != f True

Agora, só para conferir se você entendeu bem, marque somente as expressões incorretas: Tomando a = 4, b = 10, c = 5, d = 0 e f = 5.0

- [x] a == b True
- [ ] a < b True
- [x] a < b False
- [ ] c >= d True
- [x] c == f False
- [ ] a < d False
- [x] c != f True

Operadores Lógicos

Uma forma de agrupar operações com lógica booleana, é utilizando operadores lógicos. Os três operadores portados básicos pelo Python são: `not`, `and`, `or`. Eles são respectivamente a negação, conjunção e disjunção.

Operador Python	Operação
not	não
and	e
or	ou

- Operador **not**

O operador **not** é um operador unário, pois só é necessário utilizar um operando com ele.

```
>>> not True
False
>>> not False
True
```

Este operador também é conhecido como "inversão", pois o valor que é verdadeiro negado se torna falso, e vice-versa.

- Operador **and**

O operador **and** tem resultado verdadeiro apenas quando seus dois operandos forem verdadeiros.

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
```

- Operador **or**

No caso do operador **or**, o resultado será false apenas se seud dois operandos também forem falsos. Caso um deles seja verdadeiro, ou os dois fores, a resultado será verdadeiro.

```
>>> True and True
True
>>> True and False
True
>>> False and False
False
```

Variáveis String

Nem sempre variáveis numéricas e lógicas vão ser suficientes para as suas necessidades. Em algumas situações, precisaremos salvar nomes e textos, nesses casos, usararemos variáveis do tipo string que armazenam cadeias de caracteres (sequência de símbolos como letras, números, sinais de pontuações, etc).

As variáveis string vão funcionar como uma sequência de blocos, cada um enumerado e contendo uma letra, um número, um símbolo ou um espaço em branco:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	p	r	e	n	d	e	n	d	o		P	y	t	h	o	n

Quando você quiser identificar o momento em que seu texto começa e termina, é necessário utilizar aspas `"`.

```
>>> usuario = "Gabriela Cavalcante"
```

O fato de cada bloco ser enumerado, permite que possamos acessar caractere a caractere. Esse número inteiro que representará a posição de cada caracter será chamado de índice, e começará a ser contato pelo 0, como no exemplo dado no início dessa seção. Para acessar cada caracter, devemos informar o índice ou posição do caractere entre colchetes `[]`, começando do 0, até o tamanho da string menos 1.

```
linguagem = "Python"
>>> print (linguagem[0])
'P'
>>> print (linguagem[1])
'y'
>>> print (linguagem[5])
'n'
>>> print (linguagem[6])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Além disso, o tamanho da string pode ser obtido através da função `len`. Como retorno da função, teremos o número de caracteres da string.

```
>>> print (len("A"))
1
>>> print (len("AB"))
2
>>> print (len("Aprendendo Python"))
17
>>> print (len(""))
0
```

Operações com Strings

Em Python, as variáveis do tipo string suportam operações como fatiamento, concatenação e composição. Veremos como funcionam algumas dessas operações.

Concatenação

É possível concatenar o conteúdo de variáveis string utilizando o operador de adição `+`. Então `"aprendendo "` + `"python"` é igual a `"aprendendo python"`. Uma outra possibilidade para casos de concatenação de uma string várias vezes é usar o operador de multiplicação, como em `"A" * 3`, igual a `"AAA"`.

```
>>> s = "ABC"
>>> print (s + "D")
'ABCD'
>>> print (s + "E" * 4)
'ABCEEEE'
```

Composição

Algumas vezes é necessário compor mensagens com várias informações, por exemplo, se tivermos que exibir uma mensagem que mostre as vezes que um determinado usuário acessou determinada seção do site. Então teremos as seguintes variáveis `usuario`, `quantidade_de_acesso`, `secao`, e as informações guardadas nelas serão usadas para compor a seguinte mensagem: `"O usuário Felipe acessou 10 vezes a seção de Esporte`.

Bem, se formos fazer isso através da concatenação, não será muito prático. Uma opção é usar a composição de strings do Python, uma forma bem mais simples e clara:

```
"João tem %d anos" % idade
```

O símbolo `%` é usado para indicar a composição da string com o conteúdo da variável `idade`. O `%d` é chamado de marcador de posição, e indica que na posição em que foi colocado, ficará um valor inteiro. Veja os principais tipos de marcadores suportador pelo Python:

Marcador	Tipo
%d	Números inteiros
%s	Strings
%f	Números decimais

Pode acontecer de você ter que representar um número com duas posições, e caso o número só tenha um algorísmo, você quer completar com zeros a esquerda. Também é possível fazer isso, por exemplo `"%04d" %numero`. Caso você queira que o número ocupe quarto espaços (ou menos ou mais) sem ser completado com 0's, basta fazer `%4d`.

```
>>> idade = 50
>>> print("[%d]" % idade)
[22]
>>> print("[%03d]" % idade)
[022]
>>> print("[%3d]" % idade)
[ 22]
>>> print("[%-3d]" % idade)
[22 ]
```

No caso de números decimais, é possível identificar o tamanho do total de caracteres que serão representados e o

número de casas decimais. Por exemplo, em `%3.1f` o que estamos dizendo é que será impresso um número decimal com 3 posições, sendo um para a parte decimal.

```
>>> print("%3f"% 3)
3.000000
>>> print("%5.2f" % 5)
5.00
```

Agora lembando do exemplo dado logo no início: `"O usuário Felipe acessou 10 vezes a seção de Esporte` e as variáveis `usuario`, `quantidade_de_acesso`, `secao`. Como poderíamos fazer esse tipo de composição? Talvez seja só agora que você percebe o poder que a composição realmente tem.

```
>>> usuario = "Felipe"
>>> quantidade_de_acesso = 10
>>> secao = "Esporte"
>>> print ("O usuário %s acessou %d vezes a seção de %s" % (usuario, quantidade_de_acesso, secao))
```

Fatiamento

O fatiamento é extremamente útil e prático, além disso, o nome é altamente sugestivo. O fatiamento funciona com a utilização de dois índices da string, um que vai indicar quando a fatia que você quer da string inicia e o outro quando a fatia acaba. Mas é **importante** notar uma coisa, quando você define os índices, o final da fatia não está incluído. Por exemplo:

```
>>> nome = "Fernanda"
>>> print (nome[0:2])
'Fe'
>>> print (nome[1:2])
'e'
>>> print (s[0:4])
'Fern'
>>> print (s[0:8])
'Fernanda'
```

Você também pode omitir um dos índices, e assim estará indicando que a fatia ou vai até o fim ou começa desde o início. Caso ambos os índices sejam ommitidos, estaremos fazendo uma cópia das strings.

```
>>> nome = "Fernanda"
>>> print (s[:4])
'Fern'
>>> print (s[1:])
'ernanda'
>>> print (s[:])
'Fernanda'
```

Valores negativos também podem ser usados, nesse caso, eles indicarão posições a partir da direita. Como nos exemplos:

```
>>> nome = "Fernanda"
>>> print (s[-1:])
'a'
>>> print (s[-4:7])
'and'
>>> print (s[-2:-1])
'd'
```

======== O fatiamento é outro recurso importante do Python, e seu nome é extremamente sugestivo. Ele permite fatiarmos strings usando índices, que indicaram o início da fatia, e o fim. Atenção só para o índice que indica o fim da fatia, isso por que o fim da fatia não é incluido, por exemplo:

```
>>> user = "João"
>>> print (user[0:2])
Jo
```

Perceba que colocamos como índice de início "0" e de fim "2", porém no índice "2" temos o caracter "ã", que não é incluido na fatia. Também é possível omitir os índices. Caso omita o número da esquedar a fatia começará do primeiro caracter, caso a omissão seja do número da direta, a fatia irá até o ultimo caractere.

```
>>> user = "Gustavo"
>>> print (user[:2])
Gu
>>> print (user[1:])
ustavo
>>> print (user[:])
Gustavo
```

Você também pode também pode usar índecis negativos para indicar uma posição a partir da direita, por exemplo, -2 indicará o penúltimo caractere, -1 será o último.

```
>>> user = "Fernanda"
>>> print (user[-1:])
a
>>> print (user[-4:8])
anda
>>> print (user[-3:-1])
nd
```

5f263d6b77e27ea188b9989be1c51b5f3ff479d8

Entrada de dados

Você já deve ter percebido que todos nosso exemplos até agora trabalharam com valores já conhecidos. Nós mesmos definimos quais dados cada variável teria, mas em muitas situações esses valores vão mudar em função de diferentes fatores, como por exemplo, a entrada do usuário. Mas para que isso aconteça, é necessário permitimos que os valores possam ser passados durante a execução do programa.

É ai que entra a entrada de dados, aquele momento em que o programa recebe informações através de um dispositivo de entrada de dados ou de um arquivo. Uma função muito importante utilizada para solicidar dados do usuário é a função `input`. No exemplo a baixo temos uma chamada a função passando como parâmetro a mensagem que será exibida para o usuário, o retorno será o valor digitado. Vamos usar como exemplo o valor 12.

```
>>> x = input("Digite um número: ")
Digite um número: 12
>>> print(x)
12
```

Agora é importante prestar atenção para o tipo de entrada que você deseja. O `input` sempre vai retornar valores do tipo string. Então mesmo que seu usuário digite números, o resultado vai ser sempre string. Para corrigir isso, podemos usar a função `int` para converter os valores para inteiro, e a função `float` para converter para um número decimal.

```
>>> idade = int(input("Qual sua idade: "))
Qual sua idade: 20
print ("Sua idade é %f" %idade)
```

Exercícios

1. Faça um programa que peça dois números inteiros e imprima a soma desses dois números.
2. Escreva um programa que leia um valor em metros e o exiba convertido em milímetros.
3. Faça um programa que calcule o aumento de um salário. Ele deve solicitar o valor do salário e a porcentagem do aumento. Por fim, exiba o valor do aumento e do novo salário.
4. Converta uma temperatura digitada em Celsius para Fahrenheit. Observação: F = 9*C/5 + 32
5. Escreva um programa para calcular a redução do tempo de vida de um fumante. Pergunte a quantidade de cigarros fumados por dia e quantos anos ele já fumou. Considere que um fumante perde 10 minutos de vida a cada cigarro, calcule quantos dias de vida um fumante perderá. Exiba o total em dias.

Condições

Por vezes você vai se deparar com situações em que decisões precisam ser tomadas durante a execução do seu programa. Por exemplo, se um usuário logou no sistema e ele é um usuário administrador, então vai poder ter acesso aos dados dos funcionários, porém se um funcionário logar, ele terá acesso só as suas informações. Uma outra situação seria exibir uma mensagem de reprovação caso o aluno não consiga notas na média, ou então só exebir aprovado caso ele tenha passado.

Nesses casos trechos de código poderão ser executados ou não, para tratar isso necessitaremos de expressões lógicas para representar essas escolhas. Veremos a seguir tais expressões que farão o controle do fluxo do programa.

if

A ideia por trás das condições é controlar quando um trecho do código deve ser executado ou não. Você vai controlar o fluxo do seu programa. Em Python, a estrutura usada para decisões é o `if`. O seu formato é mostrado a baixo.

```
if (condição):
    # bloco de código
```

Caso a condição seja verdadeira, o bloco de código identificado dentro do `if` será executado. O `if` funcionará como um "se", então "se a condição for verdadeira, faça algo".

```
idade = int(input("Qual sua idade: "))
if idade < 12:
    print("Criança")
if idade > 12:
    print("Adolescente")
if idade > 18:
    print("Adulto")
if idade > 60:
    print("Idoso")
```

Na segunda linha do nosso exemplo, temos a condição `idade < 12`. Essa condição será avaliada, e se for verdadeira, a linha seguinte será executada, imprimindo "Criança" na tela. Caso a condição seja false, a terceira linha será ignorada, e o código seguinte ao `if` será executado.

Observe também como indicamos que um bloco estará se iniciando. Veja que na linha do `if` terminamos com o símbolo dois pontos (:). Depois dele, teremos o código referente aquela condição identado, indicando que ele só será executado caso a condição seja verdadeira.

Talvez você já tenha percebido que em nosso exemplo, dependendo da idade, a saida será um pouco estranha. Por exemplo, se alguém digitar 65 como sua idade, a saida será "Adolescente", "Adulto" e "Idoso". Porém só queríamos que saisse uma única faixa referente a idade que foi colocada como entrada. Nas próximas sessões veremos uma solução para esse caso.

else

Em algumas situações, você vai querer que apenas uma condição seja verdadeira. Em Python nós podemos usar a cláusula `else` para encaminhar o fluxo da execução caso o resultado das avaliações das condições até aquele momento tenham sido falsas. Por exemplo:

```
nota = int(input("Digite sua nota: ")
if nota < 5:
    print ("Você está reprovado")
if nota < 7:
    print ("Você está em recuperação")
else:
    print ("você foi aprovado")
```

Observe que quando usamos o `else`, também usamos o ":" logo após. Isso por que ele também inicia um bloco assim como o `if`. E não foi necessário colocar condições, pois só executaremos o que estiver dentro do `else` caso não haja `if` verdadeiro.

elif

Você pode se deparar com códigos que possuam multiplos `ifs` aninhados. Como no exemplo:

```
valor_compra = float(input("Valor da compra: "))
if valor_compra < 100:
    desconto = valor_compra * 0.10
else:
    if valor_compra < 500:
        desconto = valor_compra * 0.20
    else:
        desconto = valor_compra * 0.30
```

Pode parecer que essas várias condições aninhadas não são um grande problema, mas isso é por que o nosso exemplo é simples. No caso de grandes projetos em que existem várias condições, esse código ficaria mais complicado de se entender. Para solucionar isso, Python fornece a cláusula `elif`, substituindo um `else` ou um `if`, e evitando criar um outro nível na estrutura do código. Reescrevendo o código que fizemos no início da sessão:

```
valor_compra = float(input("Valor da compra: "))
if valor_compra < 100:
    desconto = valor_compra * 0.10
elif valor_compra < 500:
    desconto = valor_compra * 0.20
else:
    desconto = valor_compra * 0.30
```

Exercícios - Condições

1. Faça um Programa que peça os valores de três lados de um triângulo. O programa deverá informar se os valores podem ser válido para formar um triângulo. Indique, caso os lados formem um triângulo, e se o mesmo é: equilátero, isósceles ou escaleno.

   Observação: Dizemos que um triângulo é equilátero quando ele possui os três lados iguais, isóceles quando possui dois lados iguais e escaleno quando não possui nenhum dos lados iguais.

2. Determine se um ano é bissexto.

   Observação: São bissextos todos os anos divisíveis por 4, excluindo os que sejam divisíveis por 100, porém não os que sejam divisíveis por 400.

3. Faça um Programa que leia três números e mostre o maior deles.

4. Faça um Programa que leia três números e mostre o maior e o menor deles.

5. Escreva um programa para aprovar o empréstimo bancário para compra de uma casa. O programa deve perguntar o valor da casa a comprar , o salário e a quantidade de anos a pagar. O valor da prestação mensal não pode ser superior a 30% do salário. Calculo o valor da prestação como sendo o valor da casa a comprar dividido pelo número de meses a pagar.

6. **(Desafio)** João Papo-de-Pescador, homem de bem, comprou um microcomputador para controlar o rendimento diário de seu trabalho. Toda vez que ele traz um peso de peixes maior que o estabelecido pelo regulamento de pesca do estado de São Paulo (50 quilos) deve pagar uma multa de R$ 4,00 por quilo excedente. João precisa que você faça um programa que leia a variável peso (peso de peixes) e verifique se há excesso. Se houver, gravar na variável excesso e na variável multa o valor da multa que João deverá pagar. Caso contrário mostrar tais variáveis com o conteúdo ZERO.

Repetições

Você já deve ter se deparado com códigos que precisavam ser executados várias vezes em função de alguma condição. Por exemplo, um código que simula um contador, e a cada 1 hora ele exibe uma mensagem. Um outro exemplo bem mais simples seria imprimir na tela os números em ordem crescente de 0 a 10. O nosso exemplo pode parecer simples, mas e se tivéssemos que imprimir números de 0 a 100, ou 1000 registros que estão armazenados no banco de dados?

Uma das estruturas que o Python disponibiliza é o `while`. Ele vai repetir um bloco de código enquanto uma condição for verdadeira. Veja a sua estrutura a seguir:

```
while (condição):
    # bloco de código
```

O exemplo que demos para escrever números de 0 a 10 ficaria da seguinte forma:

```
x = 0
while x <= 10:
    print(x)
    x = x + 1
```

Inicialmente atribuimos 0 a variável `x`. Quando vamos para a segunda linha, o `while` define uma condição para ser iniciado, `x` precisa ser menor ou igual a 10. Caso a condição seja verdadeira, a terceira e quarta linha serão execuradas enquanto a condição continuar sendo verdadeira.

Interrompendo a repetição

Você deve ter percebido que a estrutura `while` faz somente uma verificação, que acontece no início de cada repetição. O problema é que em alguns momentos você talvez precise fazer alguma validação dentro do `while` e talvez até parar a repetição. Para isso usamos a instrução `break`, que interrompe a execução do `while`.

```
numero = 0
while True:
    numero = int(input("Digite um número positivo: ")
    if numero < 0:
        break
```

Nesse exemplo, nós temos no lugar da condição do `while` o `True`, isso fará com que o while seja executado "para sempre". Depois nós temos a requisição de uma informação que será informada pelo usuário, e depois uma validação, que verificará se o número digitado é menor que 0. Caso seja, o `break` será ativado e parará com o laço de repetição.

Repetições aninhadas

Também é possível combinar vários `while`, por exemplo, se você quiser incrementar variáveis em momentos diferentes. No exemplo que temos a seguir, vamos fazer uma tabuada de multiplicação de 1 a 10.

```
tabuada = 1
while tabuada <= 10:
    num = 1
    while num <= 10:
        print("%d x %d = %d" % (tabuada, num, tabuada * num))
        num += 1
    tabuada += 1
```

Neste exemplo, retirado do livro Introdução a Programação com Python, você pode perceber que temos um primeiro laço

que será repetido enquanto a variável `tabuada` for menor ou igual a 10. Dentro desse laço temos a variável `num` e o segundo `while`, que será repetido enquanto `num` for menor ou igual a 10. Então perceba que o segundo laço será disparado toda fez que ocorrer uma repetição do primeiro.

while

Uma das estruturas que o Python disponibiliza é o `while`. Ele vai repetir um bloco de código enquanto uma condição for verdadeira. Veja a sua estrutura a seguir:

```
while (condição):
    # bloco de código
```

O exemplo que demos para escrever números de 0 a 10 ficaria da seguinte forma:

```
x = 0
while x <= 10:
    print(x)
    x = x + 1
```

Inicialmente atribuimos 0 a variável `x`. Quando vamos para a segunda linha, o `while` define uma condição para ser iniciado, `x` precisa ser menor ou igual a 10. Caso a condição seja verdadeira, a terceira e quarta linha serão execuradas enquanto a condição continuar sendo verdadeira.

Comando `break`, `continue` e `pass`

Você deve ter percebido que a estrutura `while` faz somente uma verificação, que acontece no início de cada repetição. O problema é que em alguns momentos você talvez precise fazer alguma validação dentro do `while` e talvez até parar a repetição. Para isso usamos a instrução `break`, que interrompe a execução do `while`.

```
numero = 0
while True:
    numero = int(input("Digite um número positivo: ")
    if numero < 0:
        break
```

Nesse exemplo, nós temos no lugar da condição do `while` o `True`, isso fará com que o while seja executação "para sempre". Depois nós temos a requisição de uma informações que será informada pelo usuário, e depois uma validação, que verificará se o número digitado é menor que 0. Caso seja, o `break` será ativado e parará com o laço de repetição.

Outra instrução muito útil é o `continue`, que permite avançar para a próxima iteração do laço mais interno.

```
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]:
    if i < 5:
        print i
    else:
        continue
```

Nesse exemplo, você verá que a saida mostrará os números 0, 1, 2, 3, 4. Isso acontece por que para os números menores que 5 no intervalo de 0 a 10, ocorrerá sua impressão na tela, mas para os números maiores, a repetição pulará para a próxima iteração.

Já o comando `pass` não faz muita coisa, na verdade, sua utilidade está em não fazer nada. Ela é util em situações em que sintaticamente, é necessário um comando, mas semânticamente não é necessária nenhuma ação.

Também podemos usar o `pass` quando queremos reservar o espaço de um código que ainda vamos implementar em uma função ou bloco de repetição ou condição. Por exemplo:

```
class MinhaClasseParaImplementar:
```

```
    pass  # implementar código dessa classe
```

Repetições aninhadas

Também é possível combinar vários `while`, por exemplo, se você quiser incrementar variáveis em momentos diferentes. No exemplo que temos a seguir, vamos fazer uma tabuada de multiplicação de 1 a 10.

```
tabuada = 1
while tabuada <= 10:
    num = 1
    while num <= 10:
        print("%d x %d = %d" % (tabuada, num, tabuada * num))
        num += 1
    tabuada += 1
```

Neste exemplo, retirado do livro Introdução a Programação com Python, você pode perceber que temos um primeiro laço que será repetido enquanto a variável `tabuada` for menor ou igual a 10. Dentro desse laço temos a variável `num` e o segundo `while`, que será repetido enquanto `num` for menor ou igual a 10. Então perceba que o segundo laço será disparado toda fez que ocorrer uma repetição do primeiro.

for

Se você já entrou em contato com o `for` em outras linguagens, vai ver algumas diferenças na utilização dele em Python. Por exemplo, em C, o `for` é usado para criar uma iteração e definir uma condição de parada. Já no Python, este comando itera sobre itens de uma sequência (lista ou strings, por exemplo), seguindo a ordem em que aparecem na sequência.

```
>>> # Medir o tamanho de algumas strings:
>>> a = ['gato', 'janela', 'defenestrar']
>>> for x in a:
...     print x, len(x)
...
gato 4
janela 6
defenestrar 11
>>>
```

Exercícios - Repetições

1. Faça um programa que peça uma nota, entre zero e dez. Mostre uma mensagem caso o valor seja inválido e continue pedindo até que o usuário informe um valor válido.

2. Faça um programa que leia um nome de usuário e a sua senha e não aceite a senha igual ao nome do usuário, mostrando uma mensagem de erro e voltando a pedir as informações.

3. Supondo que a população de um país "A" seja da ordem de 80.000 habitantes com uma taxa anual de crescimento de 3% e que a po pulação de "B" seja 200.000 habitantes com uma taxa de crescimento de 1.5%. Faça um programa que calcule e escreva o número de anos necessários para que a população do país "A" ultrapasse ou iguale a população do país "B", mantidas as taxas de crescimento

4. Escreva um programa que exiba uma lista de opções (menu): adição, subtração, divisão, multiplicação e sair. Imprima a tabuada da operação escolhida. Repira até que a opção saída seja escolhida.

5. **(Desafio)** A seqüência de Fibonacci é a seguinte: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... Sua regra de formação é simples: os dois primeiros elementos são 1; a partir de então, cada elemento é a soma dos dois anteriores. Faça um algoritmo que leia um número inteiro apresente a sequência de Fibonacci até este número.

   Observação: F1 = 1, F2 = 1, F3 = 2, etc.

Estruturas de dados

Você deve lembrar que nas sessões anteriores vimos alguns tipos básicos em Python, agora nós vamos abordar alguns tipos diferentes, como listas, sets, dicionários, tuplas, etc.

Listas

Variáveis do tipo Lista, permitem armazenar vários valores e os deixam disponíveis para serem acessados através de índices. Os valores armazenados em uma lista podem ser do mesmo tipo, de diferentes tipos ou podem ser até mesmo outras listas.

```
andares = [ 0, 1, 2, 3, 4 ]
```

Neste exemplo temos a lista `andares`, com 5 valores: 0, 1, 2, 3, 4. Dizemos que ela tem 5 elementos, que podem ser acessados por seus índices, por exemplo, usando a referência `andares[0]`, poderemos acessar o valor 0, já com `andares[3]`, acessaremos o valor 3. Para modificar uma lista, basta fazer a atribuição de um novo valor a uma posição da lista, por exemplo, `andares[2] = 5`. Perceba que quando for tentar acessar o valor de `andares[2]`, será exibido 5.

Usando como fonte a Documentação Python, aqui estão todos os métodos disponíveis em um objeto lista e no fim, o exemplo dado na documentação:

`list.append(x)` : Adiciona um item ao fim da lista; equivale a `a[len(a):] = [x]`. É importante saber que ainda existe outra forma de adicionar elementos a uma lista, `a.append(1)` equivale a `a+[1]`. Esteja atento a utilização do `append` com uma lista sendo passada por parâmetro, `a.append([3, 4])`, em situações como essa você notará que a lista será inteiramente adicionada, como se fosse um novo elemento, ficaria por exemlpo: `[ 1, 2, [3, 4]]`.

`list.extend(L)` : Prolonga a lista, adicionando no fim todos os elementos da lista L passada como argumento; equivalente a `a[len(a):] = L`. Mesmo que o `extend` só aceite listas, ele vai ter um resultado diferente de quando usamos o `append` com lista como parâmetros. O `extend` vai "extrair" os elementos passados por parâmetro e inserí-los em nossa lista.

`list.insert(i, x)` : Insere um item em uma posição especificada. O primeiro argumento é o índice do elemento antes do qual será feita a inserção, assim `a.insert(0, x)` insere no início da lista, e `a.insert(len(a), x)` equivale a `a.append(x)`.

`list.remove(x)` : Remove o primeiro item encontrado na lista cujo valor é igual a x. Se não existir valor igual, uma exceção ValueError é levantada.

`list.pop([i])` : Remove o item na posição dada e o devolve. Se nenhum índice for especificado, `a.pop()` remove e devolve o último item na lista. (Os colchetes ao redor do i indicam que o parâmetro é opcional, não que você deva digitá-los daquela maneira. Você verá essa notação com frequência na Referência da Biblioteca Python.)

`list.index(x)` : Devolve o índice do primeiro item cujo valor é igual a x, gerando ValueError se este valor não existe

`list.count(x)` : Devolve o número de vezes que o valor x aparece na lista.

`list.sort()` : Ordena os itens na própria lista in place.

`list.reverse()` : Inverte a ordem dos elementos na lista in place (sem gerar uma nova lista).

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
```

```
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Usando listas como filas

Você também pode usar uma lista como uma fila, mas para isso ela deve obedecer algumas regras com relação a inclusão e remoção de um elemento. No caso das filas, o primeiro item adicionado é o primeiro a ser recuperado (“primeiro a entrar, primeiro a sair”, FIFO, "first in first out"), semelhante as filas que fazemos em nosso dia a dia.

Para que uma lista funcione como fila, podemos trabalhar de duas formas diferentes. Se imaginarmos uma fila de clientes em uma loja, quando um cliente chegar, basta realizar um `append` para ele ser inserido na fila (`fila.append(cliente)`). Já para retirar um cliente da fila, podemos usar o método `pop` (`fila.pop(0)`), que irá retornar o valor e depois excluí-lo.

Usando listas como pilhas

Os métodos que já temos para a lista fazendo com que seja muito fácil utilizar listas como pilhas. Sua política funciona basicamente com a adição de um item no topo e a remoção também é feita no topo (política “último a entrar, primeiro a sair”, LIFO, "Last in First Out). Para adicionar um item ao topo da pilha, use append(). Para recuperar um item do topo da pilha use pop() sem nenhum índice.

Comando `del`

Uma outra forma de remover um item de uma lista conhecendo apenas seu índice, ao invés de seu valor é utilizando o comando `del`. Ele será diferente do método `list.pop()` pois devolve o item removido. O comando `del` também pode ser utilizado para remover partes (fatias) da lista, ou até a lista toda. Um bom exemplo dados na Documentação Python é este:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

Quando a remoção acontece, note que os índices são reorganizados. `del` também pode ser usado para remover totalmente uma variável:

```
>>> del a
>>> a
Traceback (most recent call last):
  ...
NameError: name 'a' is not defined
```

Caso você tente referenciar a variável `a` depois de sua remoção, esta ação vai gerar um erro (pelo menos até que seja feita uma nova atribuição para ela).

Tamanho da lista

Uma informação importânte quando se trabalha com listas é o seu tamanho. Usando a função `len` teremos como retorno o número de elementos de nossa lista.

```
>>> lista = [ 1, 2, 3 ]
>>> len(lista)
3
```

```
>>> l = []
>>> len(l)
0
```

Usando o `range`

A função `range` com certeza será de grande ajuda em algum momento da sua vida. Basicamente, ela pode gerar listas simples de forma mais simples ainda. Por exemplo:

```
for num in range(10):
    print (num)
```

Se você executar esse código, vai perceber que a função `range` gerou números de 0 a 9. Por padrão, ao passar o parâmetro único, como no exemplo, o `range` vai gerar números iniciando no zero, até o anterior do parâmetro. Mas nós também podemos indicar qual é o primeiro número que será gerado, basta passar por parâmetro o número de início e de fim.

```
for num in range(5, 10):
    print (num)
```

Ainda é possível passar um terceiro argumento, que será o intervalor do salto entre cada número gerado. Por exemplo:

```
for num in range(0, 20, 2):
    print (num)
```

A saida desse código serão os números: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18. O primeiro número do parâmetro diz a partir de onde o intervalo começará, o segundo número refere-se ao fim do interlado, e o terceiro número ao valor do "pulo" entre os números gerados pelo `range`.

Tuplas

Vamos ver agora um novo tipo de sequência, um outro tipo de sequência padrão na linguagem: a tupla (tuple). Uma tupla consiste em uma sequência de valores separados por vírgulas, podendo ser vista como lista em Python, com a diferença de ser imutável (assim como strings).

```
>>> t = 1, 2, 'python'
>>> t[0]
1
>>> t
(1, 2, 'python')
>>> # Podemos aninhar tuplas:
... u = t, (1, 2, 3, 4, 5)
>>> u
((1, 2, 'python'), (1, 2, 3, 4, 5))
```

No exemplo retirado da Documentação Oficial, as tuplas são sempre envolvidas por parênteses quando aparecem na saída, a vantagem disso é que quando houverem tuplas aninhadas, a sua leitura será feita corretamente.

Uma das grandes utilidades das Tuplas é para a representação de valores constante, alem disso elas podem ser usadas de diversas formas: pares ordenados (x, y), registros de funcionário extraídos uma base de dados, etc.

Um ponto interessante é a criação de tuplas contendo 0 ou 1 itens: a sintaxe usa certos truques para acomodar estes casos. No caso das Tuplas vazias, um par de parênteses vazios é o necessário para construí-la; uma tupla unitária é construída por um único valor e uma vírgula entre parênteses (não basta colocar um único valor entre parênteses). Um pouco estranho, mas é assim que funciona:

```
>>> vazia = ()
>>> upla = 'hello',    # aqui está a vírgula no final
>>> len(vazia)
0
>>> len(upla)
1
>>> upla
('hello',)
```

As tuplas também suportam acesso aos valores através dos índices e maior parte das operações das listas, como fatiamento. Podemos utilizá-las também com o `for`:

```
>>> numeros = (1, 2, 3, 4)
>>> for num in numeros:
        print (num)
```

Python também permite operações chamadas de empacotamento e desempacotamento. O empacotamento acontece como no exemplo: `t = 12345, 54321, 'python!'` . Os valores 12345, 54321, 'python!' são empacotados na tupla `t` . Já o desempacotamento acontece como neste caso: `a, b = 10, 20` . Para funcionar, é necessário que a lista de variáveis do lado esquerdo tenha o mesmo comprimento da sequência à direita. Sendo assim, a atribuição múltipla é um caso de empacotamento de tupla e desempacotamento de sequência. Ainda é possível fazer trocas rapidamento dos valores das variáveis:

```
>>> a, b = b, a  # troca os valores de a e b
```

Também podemos criar tuplas a apartir de listas, usando a função `tupla` .

```
>>> l = [1, 2, 3]
>>> t = tuple(l)
```

```
>>> t
(1, 2, 3)
```

Mesmo que não possamos fazer alterações na tupla depois de criá-la, podemos concatená-las... mas saiba que isso gera novas tuplas:

```
>>> t1 = (1, 2, 3)
>>> t2 = (4, 5, 6)
>>> t1 + t2
(1, 2, 3, 4, 5, 6)
```

IMPORTANTE: Tuplas podem conter objetos que podem ser alterados, mas as alterações nesses objetos não são consideradas mudanças na tupla em si, como no exemplo:

```
>>> tupla = ("a", ["b", "c", "d"])
>>> tupla
('a', ['b', 'c', 'd'])
>>> len(tupla)
2
>>> tupla[1]
['b', 'c', 'd']
>>> tupla[1].append("e")
>>> tupla
('a', ['b', 'c', 'd', 'e'])
```

Set

Python também inclui um tipo de dados para conjuntos, chamado `set`. Um conjunto é uma coleção desordenada de elementos, mas não possui elementos repetidos. A vantagem desse tipo de dado é quando precisamos garantir que dados não se repitam, funcionando como um verificação eficiente da existência de objetos e a eliminação de itens duplicados. Conjuntos também suportam operações matemáticas como união, interseção, diferença e diferença simétrica.

Uma pequena demonstração que tem como fonte a Documentação Python:

```
>>> cesta = ['uva', 'laranja', 'uva', 'abacaxi', 'laranja', 'banana']
>>> frutas = set(cesta)   # criar um conjunto sem duplicatas
>>> frutas
set(['abacaxi', 'uva', 'laranja', 'banana'])
>>> 'laranja' in frutas   # testar se um elemento existe é muito rápido
True
>>> 'capim' in frutas
False
>>>
>>> # Demonstrar operaçes de conjunto em letras únicas de duas palavras
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                 # letras unicas em a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                             # letras em a mas não em b
set(['r', 'd', 'b'])
>>> a | b                             # letras em a ou em b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                             # letras tanto em a como em b
set(['a', 'c'])
>>> a ^ b                             # letras em a ou b mas não em ambos
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Dicionário

Outra estrutura de dados muito útil embutida em Python é o dicionário, cujo tipo é `dict`. É uma estrutura parecida com a lista, mas possui propriedades de acesso diferentes. Dicionários são delimitados por chaves: {}, e contém uma lista de pares chave:valor separada por vírgulas. Eles são também chamados de “memória associativa” ou “vetor associativo” em outras linguagens.

Diferente das listas, tuplas, e outras sequências indexadas por inteiros, dicionários são indexados por chaves (keys), que podem ser de qualquer tipo imutável (como strings). Tuplas também podem ser chaves se contiverem apenas strings, inteiros ou outras tuplas. Se a tupla contiver, direta ou indiretamente, qualquer valor mutável, não poderá ser chave. Listas não podem ser usadas como chaves porque podem ser modificadas in place pela atribuição em índices ou fatias, e por métodos como append() e extend() [PyDoc].

Uma forma de visualizar como dicionários funcionam é vê-los como um conjunto não ordenado de pares chave-valor. Essas chaves são únicas em uma dada instância do dicionário, caso um valor que já foi atribuido a uma chave seja usado novamente, o valor associado é alterado para um novo valor, caso não existe, o par chave:valor será adicionado no dicionário.

Veja a tabela a seguir e como ela pode ser representada como um dicionário:

Produto	Valor
Banana	3.00
Maçã	5.00
Uva	8.00

```
mercadinho = { "Banana" : 3.00,
               "Maçã"   : 5.00,
               "Uva"    : 8.00 }
```

Para acessar as informações do dicionário, devemos utilizar duas chaves. Usando o exemplo que demos, para acessar o preço da banana devemos usar `mercadinho["Banana"]`. Além da operação de armazenar, outra operação muito importante é a de recuperar valores a partir de chaves. Também é possível remover um par chave:valor com o comando `del`. Se você armazenar um valor utilizando uma chave já presente, o antigo valor será substituído pelo novo. Se tentar recuperar um valor usando uma chave inexistente, será gerado um erro do tipo `KeyError`. Como no exemplo:

```
>>> mercadinho = { "Banana" : 3.00,
...                "Maçã"   : 5.00,
...                "Uva"    : 8.00 }
>>> print(mercadinho["Manga"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Manga'
>>>
```

Se quiser verificar se existe a chave existe no dicionário, use o operador `in`.

```
>>> mercadinho = { "Banana" : 3.00,
...                "Maçã"   : 5.00,
...                "Uva"    : 8.00 }
>>> print("Manga" in mercadinho)
False
>>> print("Uva" in mercadinho)
True
>>>
```

Dois métodos também muito importantes são o `keys()` e o `values()`. O `keys()` devolve a lista de todas as chaves presentes no dicionário, em ordem arbitrária (se desejar ordená-las basta aplicar o a função sorted() à lista devolvida). Já o `values()` irá retornar os valores do dicionários. O retorno de ambos os métodos são geradores, então você pode usá-los dentro de um `for` ou transformá-lo em lista com a função `list`.

A seguir, um exemplo de uso do dicionário retirado do PyDoc:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

O construtor dict() produz dicionários diretamente a partir de uma lista de chaves-valores, armazenadas como duplas (tuplas de 2 elementos). Quando os pares formam um padrão, uma list comprehensions pode especificar a lista de chaves-valores de forma mais compacta.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use uma list comprehension
{2: 4, 4: 16, 6: 36}
```

Quando chaves são strings simples, é mais fácil especificar os pares usando argumentos nomeados no construtor:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Caso você tenha ficado em dúvida em que situação usar dicionários ou listas, aqui vai uma dica. Se seus dados não precisam ser acessados de uma só vez, e se você normalmente usa chaves acessá-los... um dicionário seria mais interessante. Um outro ponto é a possibilidade de acessar valores através de uma chave rapidamente sem precisar fazer pesquisas, a implementação interna do docionário permite uma boa velocidade de acesso quando temos muitas chaves. Porém se você quiser manter a ordem de inserção, use listas. O dicionário não organizar as chaves, ou seja, quando você insere a primeira chave, não significa que ela manterá a posição de primeira quando você adicionar outras.

Exercícios - Listas

1. Faça um programa que leia uma expressão com parênteses. Usando pilhas, verifique se os parênteses foram abertos e fechados na orgem correta. Exemplo:

   (( )) - OK

   (( )) (( ) ( )) - OK

   ( )) - Erro

Exercícios - Dicionário

1- Crie um dicionário e armazene nele os seus dados: nome, idade, telefone, endereço. Imprima todos os dados usando o padrão chave: valor.

2 – Crie um programa que, usando dicionário, crie uma agenda de tamanho fornecido inicialmente pelo usuário. Leia os dados de todos os contatos do usuário de forma que a agenda fique completa e por fim imprima todos os contatos.

Um pouco mais sobre Strings

Vimos logo no início do material algumas informações sobre strings, mas existem bem mais informações sobre elas do que você imagina. Algo que deve estar lembrado é que strings são imutáveis:

```
>>> linguagem = "python"
>>> linguagem[0] = "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Mas e se você quiser trabalhar com caractere a caractere? Transforme a string em lista:

```
>>> linguagem = list("python")
>>> linguagem
['p', 'y', 't', 'h', 'o', 'n']
>>> linguagem[0] = 'a'
>>> linguagem
['a', 'y', 't', 'h', 'o', 'n']
>>> linguagem = "".join(linguagem)
>>> linguagem
'python'
```

Python também facilita algumas verificações com strings. Por exemplo, se você quiser saber se uma string começa ou termina com algum caractere, é possível usar os métodos `startswith` (verifica os primeiros caracteres) e `endswith` (verifica os últimos). Os métodoes retornam `True`, caso tenha encontrado os caracteres ou `False`, caso contrário.

```
>>> linguagem = "Python"
>>> linguagem.startswith("P")
True
>>> linguagem.startswith("Pyth")
True
>>> linguagem.startswith("py")
False
>>> linguagem.startswith("N")
False
>>> linguagem.endswith("hon")
True
```

Note que estes métodos tratam de forma diferente letras maiúsculas e minúsculas. Um jeito de resolver esse problema é conver a string para maiúscula ou minúscula antes de fazer a verificação. Os métodos disponíveis para fazer essa operação é o `lower` (retorna uma código com todos os caracteres minúsculos), e `upper` (retorna uma cópia com todos os caracteres maiúsculos).

```
>>> linguagem = "Python"
>>> linguagem.lower()
'python'
>>> linguagem.upper()
'PYTHON'
>>> linguagem.lower().startswith("pyt")
True
>>> linguagem.upper().startswith("PYT")
True
```

Em Python, também é possível verificar se uma string está contida em outra usando `in`, ou verificar o contrário com `not in`. Por exemplo:

```
>>> linguagem = "Estudando Python"
>>> "Est" in linguagem
```

```
True
>>> "Python" in linguagem
True
>>> "python" in linguagem
False
SyntaxError: invalid syntax
>>> "casa" not in linguagem
True
```

Também podemos fazer a contagem de ocorrências de uma letra ou palavra em uma string, para isso basta usar `count`.

```
>>> linguagem = "Estudando Python"
>>> linguagem.count("a")
1
>>> linguagem.count("n")
2
>>> linguagem.count("o")
2
```

Em situações em que queremos saber se uma string está dentro da outra e a partir de que posição ela se encontra, podemos usar o método `find`.

```
>>> s = "Aprendendo python nas férias"
>>> s.find("py")
11
>>> s.find("n")
4
>>> s.find("nas")
18
>>> s.find("w")
-1 # valor negativo caso a string não seja encontrada
```

Se quiser fazer a pesquisa da direita para a esquedar, pode usar `rfind`:

```
>>> s = "Aprendendo python nas férias"
>>> s.rfind("python")
11
>>> s.rfind("s")
28
>>> s.find("s")
20
>>> s.find("python")
11
```

Você também pode passar mais dois argumentos como parâmetro tanto de `find` como `rfind`, uma parâmetro para indicar a posição inicial e para a posição final da pesquisa.

```
>>> s = "Aprendendo python nas férias"
>>> s.find("python", 9)
11
>>> s.find("a")
19
>>> s.find("a", 20)
27
>>> s.find("a", 20, 21)
-1
```

Aqui estão mais alguns métodos que podem ser muito interessantes de serem usados durante o desenvolvimento do seu código:

`center`: centraliza a string em número de posições passadas como parâmetro, preenchedo com espaço tanto a esqueda quando a direita. É possível passa o tamanho e o caractere de preenchimento.

```
>>> s = "python"
>>> print("X"+s.center(10)+"X")
X  python  X
>>> print("X"+s.center(10, ".")+"X") #
X..python..X
```

`ljust e rjust`: centraliza a string, preenchendo com espaços à esqueda (`ljust`) ou a direita (`just`).

```
>>> s = "python"
>>> s.ljust(20)
'python              '
>>> s.rjust(20)
'              python'
>>> s.ljust(20, ".")
'python..............'
>>> s.rjust(20, ".")
'..............python'
```

`split` : quebra uma string a parti de de um caractere passado como parâmetro. O retorno é uma lista com substrings separadas. Observe que o caractere usado para dividir a string, não é retornado na lista.

```
>>> s = "o rato roeu a roupa do rei de roma"
>>> s.split(" ")
['o', 'rato', 'roeu', 'a', 'roupa', 'do', 'rei', 'de', 'roma']
>>> s.split("o")
['', ' rat', ' r', 'eu a r', 'upa d', ' rei de r', 'ma']
```

`splitlines` : separa uma string com várias linhas de texto.

```
>>> s = "primeira linha \n segunda linha \n terceira linha"
>>> s
'primeira linha \n segunda linha \n terceira linha'
>>> print(s)
primeira linha
 segunda linha
 terceira linha
>>> s.splitlines()
['primeira linha ', ' segunda linha ', ' terceira linha']
```

`replace` : substitui trechos de uma string por outros. O primeiro parâmetro é a string a substituir e o segundo, o que conteúdo que a substituirá. Ainda podemos passar um terceiro parâmetro, que irá limitar quantas vezes queremos realizar a repetição.

```
>>> s = "o rato roeu a roupa do rei de roma"
>>> s.replace("rato", "roedor")
'o roedor roeu a roupa do rei de roma'
>>> s = "um elefante, dois elefantes, três elefantes"
>>> s.replace("elefante", "canguru", 1)
'um canguru, dois elefantes, três elefantes'
>>> s.replace("elefante", "canguru")
'um canguru, dois cangurus, três cangurus'
>>> s.replace("elefante", "")
'um , dois s, três s'
>>> s.replace("", "-")
'-u-m- -e-l-e-f-a-n-t-e-,- -d-o-i-s- -e-l-e-f-a-n-t-e-s-,- -t-r-\xc3-\xaa-s- -e-l-e-f-a-n-t-e-s-'
```

`strinp` remove espaços em branco do início ou fim da string. Para remover apenas dos caracteres em branco à esqueda, podemos usar o `lstrip` , e para remover apenas os da direita `rstrip` . Você também pode passar um parâmetro para esses métodos, e então ele será removido da string.

```
>>> s = "          python     "
>>> s.strip()
'python'
```

```
>>> s.lstrip()
'python     '
>>> s.rstrip()
'           python'
>>> s = "___Python___"
>>> s.strip("_")
'Python'
>>> s.lstrip("_")
'Python___'
>>> s.rstrip("_")
'___Python'
```

`isalnum` : retorna verdadeiro se a string não estiver vazia, e se todos os caracteres forem letras e/ou número. Se a string tiver caracteres como vírgulas, espaços, interrogação, etc, o retorno será False.

`isalph` : retorna verdadeiro apenas se todos os caracteres forem letras.

`isdigit` : retorna verdadeiro apenas se todos os caracteres forem números e se a string não estiver vazia

`isupper e islower` : verifica se todos os caracteres de uma string são letras maiúsculas ou maiúsculas, respectivamente.

`isspace` : verifica se a string contém apenas caracteres em branco, como espaço, quabra de linhas, etc.

`isprintable` : verifica se existe algum caractere que não pode ser impresso na string, retornando False.

```
>>> "\n\t".isprintable()
False
>>> "Oi Mundo".isprintable()
True
```

`format` : auxilia na composição de string. Os números entre colchetes são uma referência os parâmetros passados ao método `format` .

```
>>> "{0} {1}".format("Oi", "Mundo")
'Oi Mundo'
>>> "{0} {1} {0}".format("Oi", "Mundo")
'Oi Mundo Oi'
```

Também é possível fazer formatação de números, o valor então vai ser impresso com a largura determinada e com zeros à esqueda para completar o tamanho.

```
>>> "{0:05}".format(5)
'00005'
```

Veja algumas outras formas de usar o `format` mostradas no livro "Introdução à Programação com Python":

```
>>> "{0:*=7}".format(32)
'*****32'
>>> "{0:*^7}".format(122)
'**122**'
>>> "{0:*<7}".format(122)
'122****'
>>> "{0:*>7}".format(122)
'****122'
>>> #separação de milhas
...
>>> "{0:10,}".format(1234)
'     1,234'
>>> "{0:10.5f}".format(1234)
'1234.00000'
>>> "{0:10.5f}".format(1234.5)
'1234.50000'
>>> "{0:10,.5f}".format(1234.5)
'1,234.50000'
```

É importante saber que quando se trabalha com formatos numéricos, devemos indicar com uma letra o formato que deve ser usado para a impressão, essa letra vai decidir como o número será exibido.

Código	Descrição
b	Binário
c	Caractere
d	Base 10
n	Base 10 local
o	Octal
x	Hexadecimal com letras minúsculas
X	Hexadecimal com letras maiúsculas

```
>>> "{:b}".format(123)
'1111011'
>>> "{:c}".format(65)
'A'
>>> "{:d}".format(65)
'65'
>>> "{:n}".format(95)
'95'
>>> "{:o}".format(195)
'303'
>>> "{:x}".format(15)
'f'
>>> "{:X}".format(15)
'F'
```

Exercícios - String

1. Conta a ocorrência da palavra "tigre" na frase "um tigre, dois tigres, três tigres", e a posição inicial de cada uma delas.
2. Escreva um programa que leia duas strings re uma terceira com os caracteres comuns às duas strings lidas. Exemplo:
   1ª String: AAACTBF
   2ª String: CBT
   Resultado: CBT
   A ordem dos caracteres da string rada não é importante, mas deve conter todas as letras comuns a ambas. <<<<<<< HEAD

3. Jogo da Velha

4. Faça um programa que peça um inteiro positivo e o mostre invertido. Ex.: 1234 gera 4321
5. Dado o texto:

   “The Python Software Foundation and the global Python community welcome and encourage participation by everyone. Our community is based on mutual respect, tolerance, and encouragement, and we are working to help each other live up to these principles. We want our community to be more diverse: whoever you are, and whatever your backgrou and, we welcome you.”

   Calcule quantas palavras possuem uma das letras “python” e que tenham mais de 4 caracteres. Não se esqueça detransformar maiúsculas para minúsculas ede remover antes os caracteres especiais.

6. Na pacata vila campestre de Ponteironuloville, todos os telefones têm 6 dígitos. A companhia telefônica estabelece as seguintes regras sobre os números:

   i. Não pode haver dois dígitos consecutivos idênticos, porque isso é chato;

   ii. A soma dos dígitos tem que ser par, porque isso é legal;

   iii. O último dígito não pode ser igual ao primei ro, porque isso dá azar.

   Então, dadas essas regras perfeitamente razoáveis, bem projetadas e maduras, quantos números de telefone na lista abaixo são válidos?

   213752 216732 221063 221545 225583 229133 230648 233222 236043 237330 239636 240138 242123 246224 249183 252936 254711 257200 257607 261424 263814 266794 268649 273050 275001 277606 278997 283331 287104 287953 289137 291591 292559 292946 295180 295566 297529 300400 304707 306931 310638 313595 318449 319021 322082 323796 326266 326880 327249 329914 334392 334575 336723 336734 338808 343269 346040 350113 353631 357154 361633 361891 364889 365746 365749 366426 369156 369444 369689 372896 374983 375223 379163 380712 385640 386777 388599 389450 390178 392943 394742 395921 398644 398832 401149 402219 405364 408088 412901 417683 422267 424767 426613 430474 433910 435054 440052 444630 447852 449116 453865 457631 461750 462985 463328 466458 469601 473108 476773 477956 481991 482422 486195 488359 489209 489388 491928 496569 496964 497901 500877 502386 502715 507617 512526 512827 513796 518232 521455 524277 528496 529345 531231 531766 535067 535183 536593 537360 539055 540582 543708 547492 550779 551595 556493 558807 559102 562050 564962 569677 570945 575447 579937 580112 580680 582458 583012 585395 586244 587393 590483 593112 593894 594293 597525 598184 600455 600953 601523 605761 608618 609198 610141 610536 612636 615233 618314 622752 626345 626632 628889 629457 629643 633673 637656 641136 644176 644973 647617 652218 657143 659902 662224 666265 668010 672480 672695 676868 677125 678315

Funções

Você deve ter percebido que usamos funções inúmeras vezes. Mas também temos a possibilidade de criar nossas próprias funções. Para definir novas funções devemos usar a instrução `def`, como no exemplo a baixo:

```
>>> def soma(a,b):
...     print(a + b)
...
>>> soma(5, 5)
10
```

A estrutura é bem simples, temos a instrução `def` seguida pelo nome da função (soma) e os parâmetros que a função irá receber. Os comandos referentes a esta função estarão depois dos dois pontos (:) e identados, como o `print(a +b)`.

As funções são úteis para empacotas uma tarefa espacífica em um trecho de código. A vontagem disso é a reutilização do código, a solução criada naquela função poderá ser usada sempre que necessária sem precisar que reescreve-la sempre.

Nós ainda podemos definir um retorno para a função, modificando a função da soma, vamos colocá-la para retornar o valor resultante.

```
>>> def soma(a,b):
...     return(a + b)
...
>>> print(soma(5, 5))
10
```

Variáveis locais e globais

Agora que começamos a criar nossas funções, vamos trabalhar com variáveis locais (ou internas) e variáveis globais (externas). A grande diferença delas é a visibilidade de cada uma terá.

Quando criamos variáveis locais em uma função, elas existiram apenas dentro daquela estrutura, somente para aquela função elas estarão disponíveis. Assim, não conseguiremos acesso a elas em um local fora da função em que foi criada. Para resolver isso, retornamos e passamos valores como parâmetro, para que seja possível a troca de valores entre diferentes trechos do código.

Já variáveis globais são definidas fora de um função, e são visíveis por todoas as outras funções. Caso um módulo esteja sendo importado, as suas variáveis globais também estarão disponível no módulo que o importou. Veja o seguinte exemplo:

```
>>> peso = 80
>>> tamanho = 1.80
>>> def imc():
...     return(peso/(tamanho**2))
...
>>> print(imc())
24.6913580247
```

Por mais que variáveis globais pareçam facilitar nossas vidas, elas devem ser usadas o mínimo possível em quando estiver escrevendo seu código. Esse cuidado deve acontecer por que elas dificultam a leitura do seu programa, já que você vai ter que procurar sua definição e conteúdo fora da função, além disso a variável pode ser alterada por várias funções, para encontrar qual o local exatado em que ela foi modificada, será um pouco complicado. Fora a legibilidade do código, ainda existe o comprometimento do encapsulamento da função, isso acontece porque a função dependerá de uma variável externa, que não é declarada dentro da função nem recebida como parâmetro.

Apesar dos motivos que nos fazem tomar cuidado com variáveis globais, elas também possuem suas vantagens.

Sugerimos que elas sejam utilizadas para armazenas valores constantes ou informações de configurações, que precisam ser acessíveis para todas as funções do programa.

Funções como parâmetros

Uma possibilidade que existe em Python é fazer a passagem de funções como parâmetros. Assim, será possível combinar várias funções para realizar uma tarefa.

```
def soma(a, b):
    return a+b
def subtracao(a, b):
    return a-b
def imprime(a, b, foper):
    print(foper(a,b))
imprime(5, 4, soma)
imprime(10, 1, subtracao)
```

Empacotamento e desempacotamento de parâmetros

Em Python você também pode empacotar parâmetros em uma lista, como no exemplo:

```
def soma(a, b):
    print(a+b)
L = [2, 3]
soma(*L)
```

Quando fazemos a chamada da função soma, passamos `*L`, esse asterisco irá indicar que queremos desempacotar a lista L utilizando seus valores como parâmetro para a função `soma`. No nosso exemplo `L[0]` será atribuido a `a` e `L[1]` será atribuido a `b`.

Função Lambda

Podemos criar funções semples, sem nome, chamadas de função lambda.

```
a = lambda z: x* 2
print(a(3))
```

Inicialmente definimos uma função lambda que recebe um parâmetro, no caso x, e que retorna o dobre desse número. Observe que tudo foi feito em apenas uma linha. A função é criada e atribuída a variável `a`.

Agora que já vimos a teoria sobre python, podemos começar a praticar um pouco. Selecionamos alguns exercícios para que vocês possam exercitar o que aprenderam durante o curso, a maioria desses exercícios foram retirados do curso Python para Zumbis, ministrado pelo professor Fernando Masanori, que é um dos grandes "evangelizadores" dessa linguagem aqui no Brasil.

Números Aleatórios e Estruturas de Dados

1. Escreva um programa que sorteie um número aleatório, peça um número ao suário e verifique se esse número é igual, maior ou menor ao número sorteado, caso o número seja igual, imprima “Parabéns você acertou!!!”, caso seja maior ou menor imprima algumas dicas como por exemplo, “é maior” e “é menor” ou “quente” e “frio”.

2. Sorteie 10 inteiros entre 1 e 100 para uma lista e descubra o maior e o menor valor, sem usar as funções max e min.

3. Faça um programa que crie dois vetores com 10 elementos aleatórios entre 1 e 100. Gere um terceiro vetor de 20 elementos, cujos valores deverão ser compostos pelos elementos intercalados dos dois outros vetores. Imprima os três vetores.

String

1. Faça um programa que peça um inteiro positivo e o mostre invertido. Ex.: 1234 gera 4321

2. Dado o texto:

   “The Python Software Foundation and the global Python community welcome and encourage participation by everyone. Our community is based on mutual respect, tolerance, and encouragement, and we are working to help each other live up to these principles. We want our community to be more diverse: whoever you are, and whatever your backgrou and, we welcome you.”

   Calcule quantas palavras possuem uma das letras “python” e que tenham mais de 4 caracteres. Não se esqueça detransformar maiúsculas para minúsculas ede remover antes os caracteres especiais.

3. Na pacata vila campestre de Ponteironuloville, todos os telefones têm 6 dígitos. A companhia telefônica estabelece as seguintes regras sobre os números:

   i. Não pode haver dois dígitos consecutivos idênticos, porque isso é chato;

   ii. A soma dos dígitos tem que ser par, porque isso é legal;

   iii. O último dígito não pode ser igual ao primei ro, porque isso dá azar.

   Então, dadas essas regras perfeitamente razoáveis, bem projetadas e maduras, quantos números de telefone na lista abaixo são válidos?

   213752 216732 221063 221545 225583 229133 230648 233222 236043 237330 239636 240138 242123 246224 249183 252936 254711 257200 257607 261424 263814 266794 268649 273050 275001 277606 278997 283331 287104 287953 289137 291591 292559 292946 295180 295566 297529 300400 304707 306931 310638 313595 318449 319021 322082 323796 326266 326880 327249 329914 334392 334575 336723 336734 338808 343269 346040 350113 353631 357154 361633 361891 364889 365746 365749 366426 369156 369444 369689 372896 374983 375223 379163 380712 385640 386777 388599 389450 390178 392943 394742 395921 398644 398832 401149 402219 405364 408088 412901 417683 422267 424767 426613 430474 433910 435054 440052 444630 447852 449116 453865 457631 461750 462985 463328 466458 469601 473108 476773 477956 481991 482422 486195 488359 489209 489388 491928 496569 496964 497901 500877 502386 502715 507617 512526 512827 513796 518232 521455 524277 528496 529345 531231 531766 535067 535183 536593 537360 539055 540582 543708 547492 550779 551595 556493 558807 559102 562050 564962 569677 570945 575447 579937 580112 580680 582458 583012 585395 586244 587393 590483 593112 593894 594293 597525 598184 600455 600953 601523 605761 608618 609198 610141 610536 612636 615233 618314 622752 626345 626632 628889 629457 629643 633673 637656 641136 644176 644973 647617 652218 657143 659902 662224 666265 668010 672480 672695 676868 677125 678315