Álan Crístoffer e Sousa

acristoffers@gmail.com

Curso Básico de Python 3

Plataformas de Baixo Custo Para Controle de Processos

PIBIC — FAPEMIG

Orientador: Prof. Dr. Valter Junior de Souza Leite

Laboratório de Sinais e Sistemas

CEFET/MG

Divinópolis

2017

Sumário

Capítulo 1

Introdução

Python é uma linguagem genérica, feita para ser utilizada em várias situações. Pode ser aplicada no desenvolvimento *web* (Django, Bottle), programação matemática e computacional (Orange, SymPy, NumPy), aplicativos para desktop (PyQt, PyWidgets), jogos (Pygame, Panda3D), entre outras.

A sintaxe da linguagem é simples e o código é normalmente curto. É uma linguagem que te permite focar no problema sem ficar se preocupando com a sintaxe da linguagem. Isso se torna possível pela sua grande biblioteca padrão, que contém mais de 260 módulos divididos em 35 categorias.

A linguagem é *open-source* e pode ser utilizada tanto para aplicações *open-source* quanto comerciais. A comunidade é grande e ativa, sendo fácil de encontrar suporte. Atualmente há 113700 pacotes no gerenciador de pacotes *pip*, variando desde bibliotecas de interface web a bibliotecas de matemática computacional, interfaces gráficas, visão computacional, etc.

Python é uma linguagem relativamente velha, desenvolvida nos anos 80 e publicada pela primeira vez em fevereiro de 1991. Ela foi desenvolvida por Guido van Rossum, que trabalhava no grupo de sistema operacional distribuído *Amoeba*. Ele queria uma linguagem que fosse parecida com a linguagem *ABC* e que pudesse acessar as chamadas do sistema *Amoeba*. Para isso ele resolveu criar uma linguagem extensível.

O nome Python não vem da cobra, mas sim de um grupo de comediantes dos anos 70: *Monty Python*. Guido era fã de seu show *Monty Python's Flying Circus*.

1.1 Instalação

Há várias maneiras de se obter o interpretador Python. Ele pode ser baixado diretamente do site python.org, instalado pelo gerenciador de pacotes do seu sistema operacional (apt, yum, brew, etc), instalado por gerenciadores de versão (pyenv) ou por uma suíte (Anaconda).

Este último é necessário na plataforma Windows já que não há binários para esta plataforma no repositório *PyPi* e a compilação é extremamente difícil devido a falta de suporte das bibliotecas usadas pela suíte *SciPy*.

Seguem as maneiras mais fáceis de se instalar no Windows, Ubuntu e macOS:

- **Windows** Baixe e instale o *bundle* Anaconda (`https://www.continuum.io/downloads`, também disponível para outras plataformas).
- **Ubuntu** `sudo apt install python3-scipy python3-matplotlib`
- **macOS** Instale o Homebrew: `https://brew.sh`
  `brew install python3`
  `pip3 install numpy matplotlib`

1.2 Características

Python é uma linguagem que preza a simplicidade. Ao contrário da filosofia da linguagem *Perl*, que diz haver várias maneiras de se resolver um problema, em Python normalmente há apenas uma maneira de resolver um problema. Isso a torna uma linguagem fácil de aprender e de usar no ensino de programação, já que as soluções mostradas tendem a ser as mesmas para as mesmas classes de problemas.

Por ser uma linguagem open-source e portável ela pode ser utilizada em qualquer plataforma com qualquer finalidade, mesmo comercial. Isso faz com que aprendê-la tenha mais sentido do que aprender, por exemplo, *C#* ou *Swift*, que só são realmente usadas em algumas plataformas.

A linguagem foi feita com a ideia de ser extensível, ou seja, de poder chamar funções escritas em *C* para interagir com o sistema operacional diretamente. Isso também permite que bibliotecas como a NumPy (escrita em C e Fortran) sejam possíveis e que o código

possa ser escrito em Python e tenha apenas algumas partes específicas escritas em *C*, para melhorar o desempenho, por exemplo.

Ela também é uma linguagem embarcável e pode ser utilizada como linguagem de script para facilitar o desenvolvimento de aplicativos e plugins que são feitos primeiramente em outras linguagens. Isso é comum em jogos e aplicativos que aceitam plugins para extender suas funcionalidades.

Por ser uma linguagem de alto nível ela é capaz de fazer coisas complicadas de forma simples, com poucas linhas de código. Isso torna o código menor, mais simples e fácil de manter além de agilizar o desenvolvimento. Ela é uma linguagem altamente dinâmica, o que só é possível graças a sua natureza interpretada.

Desde o início a linguagem foi desenvolvida para ser orientada a objetos, o que faz com que utilizar construções de *OOP* nela seja extremamente simples. Além disso, por ser uma linguagem genérica, ela possui uma vasta biblioteca padrão com funcionalidades que variam de computação numérica até comunicação por rede e criação de interfaces gráficas. E se não houver uma biblioteca padrão para a funcionalidade buscada, provavelmente haverão bibliotecas de terceiros disponíveis no repositório *PyPi*.

A linguagem tem uma forte filosofia seguida pela comunidade e resumida no texto *The Zen of Python*, que também pode ser obtido usando o comando import this no interpretador. Uma versão comentada com exemplos pode ser encontrada no endereço `http://artifex.org/~hblanks/talks/2011/pep20_by_example.html`.

> The Zen of Python, by Tim Peters
>
> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one– and preferably only one –obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea – let's do more of those!

1.3 Aplicações

Por ser genérica, a linguagem Python pode ser aplicada em vários cenários. Ela é comumente utilizada no desenvolvimento web e aplicativos internet, sendo a linguagem por trás de sites como o da *Mozilla*, *Reddit* e *Instagram* e aplicativos como o *SickBeard* e *CouchPotato*.

Graças a sua extensibilidade é possível utilizar *toolkits* escritas em *C* e *C++* em Python, como a WxWidgets e Qt. Aplicativos com interfaces gráficas ricas podem ser desenvolvidas dessa meneira. Exemplos são a bibliteca digital *Calibre*, o gerenciador de torrents *QTorrent*, e a *IDE*[1] *Spyder*.

Python também pode ser utilizada para resolver problemas matemáticos complexos, muitas vezes de forma simples em comparação com outras linguagens. A suite *SciPy* traz várias bibliotecas para computação numérica e científica de alta qualidade, que podem ser utilizadas não apenas em scripts para processamento de dados ou computação intensiva, mas também para a criação de aplicativos que utilizem desses conceitos como meio (inteligência artificial, deep learning, etc).

Também pode-se utilizar a linguagem Python para extender as funcionalidades de aplicativos através de plugins. *Plugins* são *scripts* que fornecem alguma funcionalidade não oferecida pelo aplicativo e que é capaz de interagir com o mesmo. Alguns aplicativos que usam Python como linguagem de script são *Abaqus* (elementos finitos), *Scribus* (ferramenta de *publishing*), *Battlefield 2*, *Civilization IV* e *Blender* (Editor 3D).

[1] *Integrated Development Environment*, ambiente de desenvolvimento integrado

Sua natureza dinâmica e facilidade de uso permitem que mudanças sejam feitas de forma simples e rápida, o que faz com que a linguagem seja ótima para prototipagem. Também é comum seu uso por times de competição de robótica, já que fica fácil alterar a estratégia do time entre jogos. Essa facilidade também a torna boa para o ensino de programação.

1.4 Linguagem Compilada vs Linguagem Interpretada

Linguagen de programação podem ser interpretadas ou compiladas. Compilação é o processo de transformar o código fonte, escrito em uma linguagem de alto nível, em um código binário específico de uma arquitetura de processador ou microcontrolador e/ou sistema operacional, real ou simulado. Esse código é chamado de código de máquina se for compilado para um hardware físico, existente e *bytecode* se for compilado para ser executado em uma máquina virtual. A Figura 1 mostra os processos visualmente.

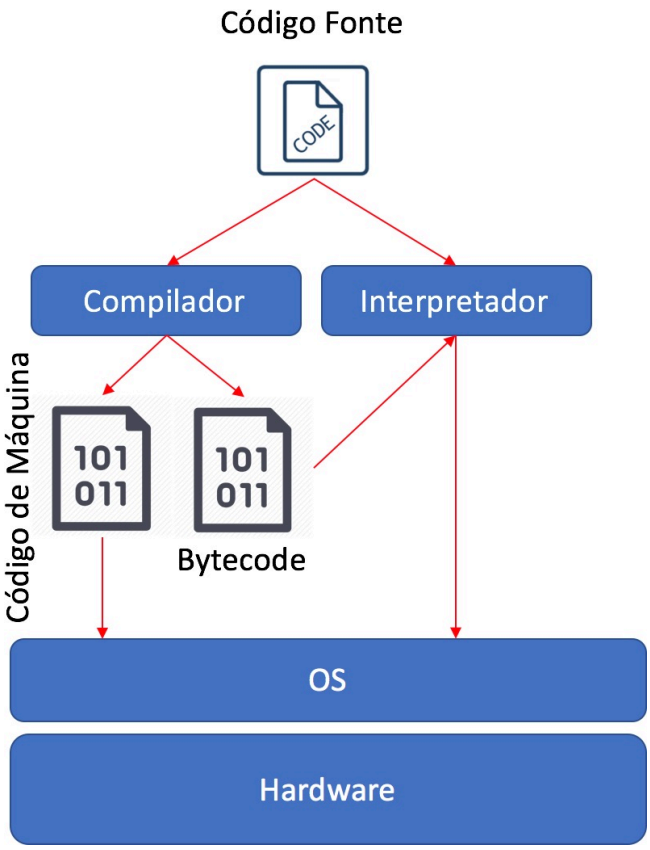

Figura 1 – Compilado vs Interpretado

O código de máquina roda diretamente no hardware, sendo executado pelo processador ou microcontrolador e possivelmente tendo sua execução coordenada pelo sistema operacional, caso haja um. O *bytecode* será executado dentro de um ambiente virtual, chamado normalmente de máquina virtual. Pode-se pensar na máquina virtual como uma forma de emulador, com a diferença que emuladores são feitos para simular um hardware existente enquanto a máquina virtual não tem a finalidade de ser um "hardware neutro", permitindo que o mesmo *bytecode* seja executado em vários hardwares diferentes.

O código interpretado é executado diretamente do arquivo fonte. O interpretador lê o código-fonte e em alto nível e o executa, sem a necessidade de uma fase de compilação.

Ambas formas tem suas vantagens e desvantagens. Código compilado, por exemplo, pode ser mais otimizado e ter vários tipos de erros testados durante a compilação. Por outro lado código interpretado costuma ser mais dinâmico e permitir estruturas e ações que são difíceis de se conseguir em linguagens compiladas.

Python é uma linguagem interpretada, mas o código passa por uma fase de compilação internamente (automaticamente). O *bytecode* gerado, no entanto, é praticamente uma reescrita do código-fonte e a única vantagem obtida da compilação é a checagem de sintaxe antes da execução e velocidade de execução. O interpretador Python não realiza nenhuma otimização ou checagem no código.

Alguns interpretadores alternativos oferecem funcionalidade de otimização, como compilação JIT (Just In Time) e compilação para *bytecode* da máquina virtual Java (Jython, PyPy, etc). No entanto tais interpretadores tem problemas na execução de aplicativos Python que dependam de bibliotecas nativas, como a *NumPy*.

1.5 Versões do Python

Existem duas versões em uso atualmente: 2.7 e 3.6. Isso ocorreu por uma questão de compatibilidade. Novos programas devem usar a versão 3.6, a não ser que o uso da 2.7 seja justificável (exemplo: uso de biblioteca que não foi portada para a 3.6).

O motivo de haver hoje duas versões é que a versão 3 modificou o significado de algumas estruturas e modificou várias partes da biblioteca padrão, renomeando e movendo métodos, transformando a estrutura *print* em uma função, entre outras modificações.

Por este motivo muito dos códigos escritos em Python 2 não rodam no Python 3.

Muitas vezes a migração é simples e pode até mesmo ser feita automaticamente por uma ferramenta, mas ainda assim algumas bibliotecas nunca foram portadas, e por isso a versão 2 não foi descontinuada. No entanto ela não recebe mais atualizações a não ser de segurança.

Existem vários interpretadores, sendo o CPython a versão oficial mantida pelo Guido. Recomenda-se usar a CPython por motivo de compatibilidade. Outras possibilidades são o Jython, um interpretador Python escrito em Java e PyPy, um interpretador Python escrito em Python. Ambos tem a vantagem de utilizarem *JIT*, mas a desvantagem de não funcionarem com bibliotecas como a NumPy.

1.6 Sintaxe

Python é uma linguagem com uma sintaxe simples e limpa, que preza pelas boas práticas de programação. Como toda linguagem de script ela não necessita de um método de entrada (*main*).

A indentação em Python é extremamente importante, já que ela define escopo. Linhas em branco são ignoradas pelo interpretador, mas podem ter significado em um ambiente interativo (normalmente elas terminam uma instrução de múltiplas linhas).

Um *Hello World* em Python é simplesmente:

```
1 print('Hello World!')
```

É comum ver um comentário na primeira linha de arquivos Python, conhecido como *shebang*: `#!/usr/bin/env python`. Esta linha é utilizada por sistemas operacionais baseados no Unix, como Linux e macOS, para permitir que o script seja chamado sem a necessidade de invocar o interpretador explicitamente. Na verdade ela é comum em várias linguagens de script, chegando ao ponto que linguagens de script onde o comentário não é feito com o caractére "#" irão aceitá-lo na primeira linha do arquivo por esse motivo.

Um exemplo de sintaxe mais elaborado:

```
1 import numpy as np # importa a biblioteca NumPy com o nome np
2
3 def squared_sum(A): # define uma função que retorna a soma dos quadrados
4     return A.T @ A  # dos valores de um vetor coluna A
```

```
5       return sum([x**2 for x in A]) # outra forma de fazer este cálculo
6
7   A = np.array([range(10)]) # vetor coluna: [1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]
8   print(squared_sum(A)) # escreve o resultado da função na tela, dado A
```

Em Python o caractére ";" pode ser usado para finalizar uma instrução, no entanto isso não é comum nem recomendado. Uma instrução em Python termina automaticamente quando uma quebra de linha é encontrada.

É uma boa prática manter o código com no máximo 79 caractéres por linha, o que pode impossibilitar a escrita de algumas instruções mais elaboradas. Para contornar isso é possível quebrar uma instrução em várias linhas de duas formas (além de ignorar a regra do 79 caractéres, o que pode ser feito vez ou outra):

- **Explicita:** terminando uma linha com o caractére "\" e continuando na linha de baixo.
- **Implicita:** expressões em parenteses, colchetes e chaves podem ser continuadas em uma nova linha sem a necessidade da barra invertida.

```
1   if 0 <= segundos < 60 and 0 <= minutos < 60 \
2      and 0 <= horas <= 24 and 1 <= dia <= 31 \
3      and 1 <= mes <= 12:
4          return True
5
6   meses = ['January',   'February', 'March',    'April',
7            'May',       'June',     'July',     'August',
8            'September', 'October',  'November', 'December']
```

1.7 Palavras Chaves

Palavras chaves são nomes reservados pela linguagem e não podem ser usados como nome de variáveis, funções ou qualquer tipo de identificador. Python é uma linguagem sensível a maiúsculas e minúsculas, sendo *if* diferente de *If*. A Tabela 1 contém uma lista de palavras chaves.

Observe que não há palavra chave para definir interface, mas eu irei me referir várias vezes a interfaces como **sequence** e **collection**. Eu me refiro a "contratos informais" e não a uma estrutura definida e reconhecida pela linguagem, como no caso de Java.

Tabela 1 – Palavras Chaves

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Estas interfaces são conjuntos de métodos que são esperados de um objeto para que ele possa ser usado em certas situações, como iterar usando **for**. Para entender melhor este conceito leia sobre *Duck Typing* (Sessão 6.4).

1.8 Identificadores

Identificadores são nomes dados a variáveis, classes, funções, métodos, etc.

- Eles podem conter letras minúsculas e maiúsculas, números e _.
- O primeiro caractére não pode ser um número.
- Qualquer outro símbolo é inválido.
- Palavras chaves não podem ser usadas como identificador.
- Não há limite no tamanho de um identificador.

Exemplos de identificadores válidos: MyClass, var_1, uma_funcao
Exemplos de identificadores inválidos: 1var, $myvar user@host

1.9 Indentação

A indentação é extremamente importante em Python já que ela define um bloco. Em *C*, por exemplo, blocos são definidos por { e }. Normalmente são utilizados 4 espaços para indentação, e eles são preferidos no lugar de tabulações.

```
1 if (true) {
2     int variavel_local = 1;
3     dentroDoBloco();
```

```
4 }
5 foraDoBloco();
```

Já em Python:

```
1 if True:
2     variavel_local = 1
3     dentro_do_bloco()
4 fora_do_bloco()
```

Tabulações e espaços podem ser misturados no início da linha para determinar a indentação, no entanto isso gerará um erro se eles forem usados de maneira que a indentação é definida pelo valor da tabulação em espaços (Python considera uma indentação igual a 8 espaços).

Um exemplo que ilustra este erro:

```
1 if True: # sem indentação
2     if True: # indentado com dois espaços
3         print('ok') # indentado com uma tabulação e um espaço
```

Por motivos de compatibilidade entre múltiplas plataformas e editores não é recomendado misturar tabulações e espaços em um mesmo arquivo. Use sempre espaços.

1.10 Comentários

Comentários em Python começam com # ou podem ser feitos usando a notação *docstring*. *Docstring* é uma string colocada como primeira instrução em uma função, classe, módulo ou pacote que serve de documentação para o mesmo. Ela se diferencia de uma string normal por usar 3 aspas simples ou duplas.

Comentários podem ser utilizados no final de uma linha, exceto se a linha for continuada usando \). Não há problemas em utilizar comentários no final de linhas continuadas implicitamente (listas e outras estruturas que separam os items por vírgula).

```
1 # isso é um comentário
2
3 """esta função não faz nada"""
4 def func():
5     pass
```

1.11 IO

Para interagir com o usuário, sistema de arquivos ou outros aplicativos e computadores na rede é necessário poder se comunicar com o mundo externo. Python contém funções em sua biblioteca padrão para isso. Aqui serão abordados apenas o console e o sistema de arquivos, mas trabalhar com *sockets* para comunicação através da rede e interprocessual não é muito diferente de trabalhar com o sistem de arquivos.

1.11.1 Console IO

Python fornece duas funções simples para interagir com o console: **input** e **print**.

```
1 nome = input('Digite seu nome: ')
2 print('Seu nome é %s' % nome)
```

A saída também pode ser formatada usando a função **format**.

```
1 print('{0} com {1}'.format('café','açucar'))
2 # café com açucar
3
4 print('{1} com {0}'.format('café','açucar'))
5 # açucar com café
```

A função *input* aceita os seguintes argumentos:

- **prompt** uma mensagem para ser exibida para o usuário. O cursor será posicionado imediatamente após essa mensagem.

A função *print* aceita os seguintes argumentos:

- **objects** os objetos a serem impressos. A função ___*str*___ será chamada para se obter uma *string* que será impressa no terminal.
- **sep** em caso de mais de um objeto, essa *string* será usada como separador. O valor padrão é " " (espaço).
- **end** *string* a ser impressa após todos os outros objetos. O valor padrão é "\n" (quebra de linha).

- **file** arquivo destino da impressão. O valor padrão é “sys.stdout” (terminal).
- **flush** se o cache deve ser escrito imediatamente. O valor padrão é “False”.

1.11.2 File IO

Manipular arquivos em Python é extremamente fácil. A função **open** retorna um objeto que te permite ler e escrever em um arquivo. Esta função aceita vários argumentos, mas iremos focar nos dois mais importantes: *file* e *mode*.

- **file** nome do arquivo a ser manipulado
- **mode** modo de abertura do arquivo

Existem 8 modos de abertura e alguns podem ser combinados.

- **r** abre para leitura (padrão)
- **w** abre para escrita, truncando o arquivo (!)
- **x** cria um novo arquivo e abre para escrita
- **a** abre para escrita, anexando ao final
- **b** modo binário
- **t** modo texto (padrão)
- + abre um arquivo de disco para atualização (ler e escrever)
- **U** modo de nova linha universal (descontinuado, não use)

Há diferença entre um arquivo aberto em modo texto e binário. É esperado que um arquivo de texto contenha linhas e possa ser lido linha por linha. Também espera-se que seja possível decodificar todos os bytes deste arquivo em texto usando, por exemplo, o padrão Unicode.

Um arquivo binário, por outro lado, não respeita nenhuma codificação e pode ter qualquer valor possível em seus bytes (0 a 255). Para ler um arquivo binário é necessário conhecer sua estrutura. Sem conhecer a estrutura do arquivo o que se tem é uma sequência

de números entre 0 e 255 que não fazem sentido, diferente de arquivos de texto, que podem ser decodificados e representados por letras, números e outros símbolos conhecidos.

É importante entender o que significa "conter linhas" em um arquivo de texto. Basicamente toda linha deve ser terminada com um caractére que define o fim da linha. Este caractére, nomeado EOL (*End Of Line*), pode ser também uma sequência. Aplicativos Windows costumam usar como EOL a sequência CR-LF (*Carriage Return — Line Feed*, \r\n). O resto do mundo usa apenas LF (\n).

Isto não é um problema quando lendo um arquivo em Python, pois esta diferença será levada em conta. O problema é quando se tenta abrir um arquivo gerado pelo Python (ou qualquer aplicativo de outra plataforma) em um programa Windows. Neste caso os finais de linha não serão reconhecidos e o aplicativo interpretará o arquivo como contendo apenas uma linha.

O objeto retornardo pela função **open** contém os métodos **write**, **read** e **readline** para escrita e leitura do arquivo. O método **seek** pode ser utilizado para alterar a posição do cursor no arquivo. O método **close** fecha o arquivo. Não se pode mais ler ou escrever no arquivo após ele ter sido fechado.

```
1 file = open('novo-arquivo.txt', 'x+t')
2 file.write('Hello World!')
3 file.seek(0)
4 print(file.read())
5 file.close()
```

O objeto de arquivo trabalha com um *buffer*. Isto quer dizer que ele não salva o conteúdo no arquivo imediatamente após a chamada do método **write**. A função **flush** pode ser utilizada para forçar a escrita de qualquer dado pendente. A função **close** chama a função **flush** automaticamente.

Foi dito que a função **seek** pode ser utilizada para alterar a posição do cursor. O cursor define a posição atual no arquivo. Quando se lê, por exemplo, uma linha, a função **readline** irá retornar o conteúdo partindo da posição atual do cursor até o próximo EOL, e avançar o cursor para após o EOL.

É importante tomar cuidado com a função **write**, pois ela não insere texto na posição do cursor, como acontece em editores de texto. Esta função irá *substituir* todo o conteúdo do arquivo partindo da posição do cursor até o fim deste pelo conteúdo escrito.

A função **seek** recebe dois argumentos: a quantidade a ser avançada e a referência. A quantidade é um número inteiro, que pode ser negativo. A referência é uma das 3 constantes definidas no pacote **io**:

- **os.SEEK_SET** início do arquivo (padrão)
- **os.SEEK_CUR** posição atual do cursor
- **os.SEEK_END** fim do arquivo

A função **tell** retorna a posição atual do cursor tomando como referência o início do arquivo.

O objeto *file* deve ser fechado toda vez que usado, caso contrário corremos o risco de perder informações e/ou deixar o arquivo marcado como usado, o que impossibilita o uso do arquivo por outros aplicativos em alguns sistemas operacionais.

Como esta tarefa é comum e repetitiva, a linguagem oferece uma forma simples de fechar o arquivo automaticamente: a estrutura **with**.

```
1 with open('novo-arquivo.txt', 'x+t') as file:
2     file.write('Hello World!')
3     file.seek(0)
4     print(file.read())
```

Além da vantagem de não ter que se lembrar de fechar o arquivo, esta estrutura limita o escopo de uso do objeto, tornando fácil entender seu ciclo de vida. Ela também garante que o método **close** será chamado, mesmo se um erro ocorrer durante o processamento de seu corpo.

1.12 Import

Grande parte das funcionalidades da Python estão em suas bibliotecas. Para usar uma biblioteca ela deve primeiramente ser importada:

```
1 import sys
2 import os.path
3
4 print(sys.argv)
5 print(os.path.curdir)
```

Ao importar, também pode-se mudar o nome da variável que receberá o módulo ou importar apenas um objeto. Os dois também podem ser combinados. A estrutura `from lib import object` também pode ser utilizada para importar todos os objetos da biblioteca e coloca-los no escopo atual. Para isso usa-se `from lib import *`. No entanto esta prática não é recomendada já que polui o escopo.

```
1 import numpy as np
2 from numpy import sum
3 from numpy import linalg as la
4
5 s = sum(np.arange(10))
```

1.13 Operadores

Operadores são símbolos que representam operações, normalmente aritiméticas ou lógicas. Os seguintes símbolos representam operadores em Python:

- Aritiméticos: $+, -, *, /, //, \%, **$
- Comparação: $>, <, ==, !=, >=, <=$
- Lógicos: *and, or, not*
- *Bitwise*: $\&, |, \wedge, \ , >>, <<$
- Atribuição: $=, +=, -=, *=, /=, \%=, //=, **=, \&=, |=, \wedge=, >>=, <<=$
- Especiais: *is, is not, in, not in*

Em Python os operadores são formas convenientes de chamar funções. O interpretador converte automaticamente os operadores em chamadas de função durante a leitura do código fonte. A Tabela 2 contém uma relação de operadores, suas funções equivalentes, significado comum e exemplo de uso.

As funções equivalentes podem ser definidas em um objeto para permitir o uso do operador. Por exemplo:

```
1 class diferente:
2     def __eq__(self, b):
```

```
3           return False
4
5       def __ne__(self, b):
6           return True
7
8   d = diferente()
9   d == d # False
10  d != d # True
```

Tabela 2 – Operadores

Operador	Função equivalente	Significado	Exemplo
<	___lt___	a menor que b	`a < b`
>	___gt___	a maior que b	`a > b`
==	___eq___	a igual a b	`a == b`
!=	___ne___	a diferente de b	`a != b`
<=	___le___	a menor ou igual a b	`a <= b`
>=	___ge___	a maior ou igual a b	`a >= b`
is		é o mesmo objeto	`a is b`
is not		são objetos diferentes	`a is not b`
	___abs___	valor absoluto	`abs(a)`
+	___add___	soma	`a + b`
&	___and___	bitwise E	`a & b`
//	___floordiv___	divide e arredonda para baixo	`a // b`
~	___inv___	bitwise not	`~a`
<<	___lshift___	bitwise left shift	`a << 2`
%	___mod___	módulo (resto da divisão)	`a \% 2`

*	___mul___	multiplicação	`a * 2`
@	___matmul___	multiplicação matricial	`A @ B`
-	___neg___	negação	`-a`
|	___or___	bitwise or	`A | B`
+	___pos___	positivação	`+a`
**	___pow___	potenciação	`a ** 2`
>>	___rshift___	bitwise right shift	`a >> 2`
-	___sub___	subtração	`a - 2`
/	___truediv___	divisão	`a / 2`
^	___xor___	ou exclusivo	`a ^ b`
Funcionam com sequências			
+	___concat___	concatena a e b	`a + b`
in	___contains___	se a está contido em b	`a in b`
del[]	___delitem___	remove item de índice n	`del a[n]`
[]	___getitem___	retorna o item de índice n	`a[n]`
[]=	___setitem___	define o item do índice n	`a[2] = b`
inplace			
+=	___iadd___	soma	`a += b`
&=	___iand___	E	`a &= b`
+=	___iconcat___	concatena sequências	`a += b`
//=	___ifloordiv___	divide e arredonda para baixo	`a //= b`

<<=	___ilshift___	bitwise left shift	`a <<= b`
%=	___imod___	módulo (resto da divisão)	`a \%= b`
*=	___imul___	multiplicação	`a *= b`
@=	___imatmul___	multiplicação matricial	`a @= b`
|=	___ior___	bitwise OU	`a |= b`
**=	___ipow___	potenciação	`a **= b`
>>=	___irshift___	bitwise right shift	`a >>= b`
-=	___isub___	subtração	`a -= b`
/=	___itruediv___	divisão	`a /= b`
∧ =	___ixor___	ou exclusivo	`a ^= b`

Capítulo 2

Variáveis

Variáveis são nomes que representam dados na memória programa. Imagine uma tabela. Na célula da 3ª linha e 5ª coluna tem o valor 42. Observe o tanto de informação que eu tive que te dar para te falar o valor de uma célula. E se eu quiser o mudar valor agora? Vou ter que repetir todas as informações que localizam a célula. E se eu disser que essa célula tem um nome? Vamos chamá-la de "total". total = 42. Algumas modificações foram feitas, agora total = 3. Muito mais simples, não? "total" é uma variável.

2.1 Tipagem

Em algumas linguagens variáveis tem tipo. Python, por ser **dinâmicamente** tipada, não atribui um tipo à variável. No entanto o objeto referenciado por esta variável tem um tipo. No caso da variável "total" acima, o objeto referenciado é o número 3. 3 é do tipo **int**. Mesmo a variável não tendo um tipo, é comum dizermos que ela tem o mesmo tipo que o objeto que ela referencia. Neste caso "total" é um int.

Mas como Python tem tipagem dinâmica as variáveis podem receber outros valores, de outros tipos. Por exemplo, total = 'Zero'. Agora "total" referencia uma **string**. Podemos dizer que agora total é uma string. Em linguagens como *C* e *Java*, que são estaticamente tipadas, isso não seria possível, já que não é permitido mudar o tipo da variável após sua declaração.

Outra característica das variáveis em Python é que elas são **fortemente** tipadas. Isto quer dizer que a linguagem não vai fazer muitas suposições quanto a possíveis mudanças de tipo. Por exemplo, em JavaScript a operação `1 + '1'` resulta em `'11'`. Como JavaScript

é uma linguagem fracamente tipada, ela faz a conversão implícita do inteiro 1 para uma string e usa o operador + como um operador de concatenação. Em Python isso gera um erro, já que os tipos são incompatíveis e a linguagem não fará suposições.

```
1 a = 1
2 b = 2
3 b = 'a'
4 a + b
5 b + a
```

As duas últimas linhas retornam os seguintes erros:

TypeError: unsupported operand type(s) for +: 'int' and 'str'
TypeError: must be str, not int

2.2 Escopo

A variável em Python existe dentro do escopo em que foi criada. Os escopos em Python são **local** e **global**. Classes também tem um escopo próprio, que funcionam de forma diferente, e não será abordado aqui.

Variáveis globais são aquelas definidas no nível de módulo (primeiro nível de identação do arquivo) ou utilizando a palavra chave **global**. Variáveis locais são as variáveis definidas dentro de uma função, desde que não tenham sido declaradas globais.

Um novo escopo é criado toda vez que se cria uma nova função utilizando a palavra chave **def** ou **lambda**. Estruturas como **if**, **for** e **while** não criam um novo escopo.

Os escopos são importantes pois delimitam a vida das variáveis. Toda variável será removida no final do escopo onde ela foi criada. Seu valor continuará na memória e pode até mesmo ser reutilizado pelo interpretador (números, por exemplo, são reutilizados. Algumas strings também). O objeto não referenciado por nenhuma variável será marcado para remoção da memória e removido no próximo ciclo do *Garbage Collector*.

```
1 a = 1
2 def b():
3     a = 2
4 print(a) # escreverá 1
```

Para referenciar uma variável de um escopo externo:

```
a = 1
def b():
    global a
    a = 2
print(a) # escreverá 2
```

2.3 Ciclo de vida

Python usa uma tecnologia chamada *Garbage Collector* (coletor de lixo) para gerenciar a memória do programa. Neste modelo o usuário não precisa fazer a remoção do objeto da memória manualmente. *Garbage Collector* resolve vários problemas, mas cria outros. Se você for construir aplicativos mais complicados, vale a pena aprender mais sobre o coletor, seu funcionamento, os problemas que podem aparecer e como resolve-los.

Ao se criar um objeto e referencia-lo, um contador de referências será iniciado no objeto. Toda vez que uma nova variável referenciar este objeto este contador será incrementado. Toda vez que uma variável deixar de referenciar o objeto ele será decrementado. Quando este contador chegar a zero o objeto será marcado para remoção e o coletor o fará em momento oportuno (o coletor é executado de tempo em tempo, conforme definido pelo interpretador).

```
a = 1 # objeto 1 criado, referencias: 1
b = 1 # referencias: 2
a = None # referencias: 1
b = None # referencias: 0, marcado para remoção
```

Como pode ser visto, o mesmo objeto pode ser referenciado por várias variáveis e só será removido quando nenhuma variável o referenciar. Isso significa que o ciclo de vida do objeto não está atrelado ao ciclo de vida da variável que o criou.

Toda variável será destruída no final do escopo onde ela foi criada, mas o objeto só será destruído quando não for mais referenciado. Assim o objeto pode viver além de seu escopo original:

```
a = 1
def b(c):
```

```
3       d = c + 1
4       return d
5   e = b(1) # esta variável tem o mesmo valor que a variável d,
6            # no entanto a variável d não existe mais.
7   e = None # o valor anterior não é mais referenciado e será removido.
```

2.4 Passagem Por Referência

Passagem por referência ou por valor diz respeito a o que o interpretador fará quando uma variável for passada como argumento para uma função. Existem duas opções: criar uma cópia do valor e passar esta cópia para a função ou criar uma referência para o mesmo espaço de memória e passar esta referência para a função.

Como em Python toda variável é uma referência para um objeto na memória, o interpretador irá copiar esta referência, fazendo da Python uma linguagem com passagem por referência.

No entanto a referência que é passada é uma referência para o objeto, e não para a variável. Isso é diferente, por exemplo, da passagem por referência em C++. Em C++, alterar a variável que recebeu a referência também altera a variável original. Em Python não. No entanto fique atento a modificações feitas ao objeto, pois estas obviamente serão vistas na variável que foi referenciada.

```
1   a = 1
2   def b(c):
3       c = 2
4   b(a)
5   print(a) # Escreverá 1
6
7   a = [1]
8   def b(c):
9       c[0] = 2
10  b(a)
11  print(a) # Escreverá [2]
```

O código acima demonstra essa diferença. No primeiro bloco, alterar a variável *c* não altera a variável *a*. Um novo valor é dado à variável *c*, o contador do objeto 1 é decrementado e *a* continua referenciando o objeto 1.

No segundo bloco uma **lista** contendo o valor um em seu primeiro índice é criado na variável *a* e passado para a função. A função então altera o primeiro item da lista. Como a lista referenciada pela variável *c* é a mesma referenciada pela variável *a*, a mudança será vista após o fim da função.

Capítulo 3

Estruturas de Dados

Python oferece várias estruturas para organizar/representar dados. Para números temos as classes **int**, **float** e **complex**, para texto temos a classe **str** e podemos agrupar objetos em instâncias das classes **list**, **tuple**, **dictionary**, entre outras. Várias classes especializadas podem ser encontradas na biblioteca padrão.

3.1 Números

3.1.1 Tipos Numéricos

Python suporta 3 tipos de números: inteiro, ponto flutuante e complexo. Eles são definidos pelas classes **int**, **float** e **complex**, respectivamente. A diferença entre um inteiro e um ponto flutuante é a presença de um ponto, ou seja, 1 é um inteiro e 1.0 é um ponto flutuante. Números complexos são na forma x+yj, onde x é a parte real e y a parte imaginária. Inteiros podem ter qualquer tamanho, mas **floats** só é preciso até 15 casas decimais.

```
1 inteiro = 1
2 ponto_flutuante = 4.2
3 complexo = 4 + 2j
4 type(inteiro) # <class 'int'>
5 isinstance(ponto_flutuante, float) # True
```

Não há tipos "básicos" em Python que mapeam diretamente para um espaço de memória. Todo dado deve ser representado por um objeto. Isso tem a desvantagem de performance, já que $2+2$ se torna uma chamada de função em um objeto que irá repassar

os valores para o interpretador fazer a soma em C. Mas tem a vantagem de permitir estruturas complicadas, como números de qualquer tamanho, independente da arquitetura.

É possível entrar diretamente com números binários, octais e hexadecimais utilizando os prefixos 0b/0B, 0o/0O e 0x/0X.

```
22 == 0b00010110 == 0o26 == 0x16
```

3.1.2 Conversão de Tipos

As classes **int**, **float** e **complex** podem ser usadas para fazer conversão entre tipos. Converter de **float** para **int** irá truncar o número (descartar a parte decimal).

```
int(3.2) # 3
int(3,7) # 3
float(4) # 4.0
complex('3+5j') # (3+5j)
int('42') # 42
```

As operações matemáticas também irão realizar a conversão para **float** automaticamente se um dos objetos for **int** e o outro for **float**.

```
1 + 1.0 == 2.0
1.0 * 7 == 7.0
```

3.1.3 Imprecisão Binária

```
1.1 + 2.2 == 3.3 # False!
```

Vale lembrar que todos os números são representados internamente de forma binária, o que pode gerar alguns resultados inesperados, como o acima. Isso ocorre pois o decimal é apenas aproximado do binário, e nem todo decimal tem uma representação binária precisa. O resultado de $1.1 + 2.2$ é na verdade 3.3000000000000003 devido a essa limitação numérica.

Para circular este problema podemos usar a classe **Decimal**, do pacote **decimal**, que permite números decimais de qualquer tamanho e mantém a precisão informada. Note no entanto que o argumento deve ser uma string para que a classe consiga manter a precisão. Se for passado um objeto **float** o mesmo já terá a imprecisão e esta será carregada.

```
import decimal

decimal.Decimal('1.1') + decimal.Decimal('2.2') == decimal.Decimal('3.3') # True
```

3.1.4 Pacote math

Funções e constantes matemáticas podem ser encontradas no pacote **math**.

```
import math

math.pi # 3.141592653589793
math.cos(math.pi) # -1.0
math.exp(10) # 22026.465794806718
math.log10(1000) # 3.0
math.sinh(1) # 1.1752011936438014
math.factorial(6) # 720
```

3.2 String

Uma **string** representa uma sequência de caractéres (letras, números e dígitos especiais). Em Python as strings são codificadas em Unicode (UTF-8), o que faz com que seja possível utilizar caractéres especiais, como acentos, símbolos e letras de outros alfabetos.

Python também suporta um tipo chamado **bytes**. Este tipo se parece com uma string, mas sem a codificação, ou seja, os valores guardados são os números de 0 a 255 (1 byte, mas bytes acima de 128 devem ser "escapados") ao invés de caractéres. É como uma lista de bytes.

Este tipo é normalmente utilizado por funções que enviam ou recebem dados pela rede. A api de um objeto **byte** é muito parecida com a api de um objeto **string**. As funções **encode** e **decode** podem ser utilizadas para converter entre os tipos, caso possível.

Uma característica importante das strings em Python é que elas são imutáveis. Toda manipulação de string não irá modificar o objeto sendo manipulado, mas criar um novo objeto contendo as alterações. Por este motivo não é possível alterar um caractére diretamente (`msg[0] = 'a'` gera um erro) nem excluir parte da string (`del msg[0]` também gera um erro).

3.2.1 Criação de Strings

Strings podem ser criadas usando aspas simples, duplas ou 3 aspas simples ou duplas. Estas últimas normalmente são utilizas para *docstrings* e strings de múltiplas linhas. Em Python as strings são imutáveis. Toda operação em uma string gera uma nova string.

```
1 simples = 'Hello World!'
2 duplas = "Hallo Welt!"
3 multilinha = """Bem
4 vindo
5 ao Python"""
6
7 byte = b'byte'
8 string = byte.decode()
9 byte = string.encode()
```

Não há diferença entre aspas duplas e simples como ocorre em outras linguagens. Para criar uma string sem interpretar sequências especiais pode-se utilizar o prefix r ou R (*raw*): `r'\n'` gera a string "\n" e não uma quebra de linha. Nota: no console interativo as barras invertidas aparecem duplicadas.

O prefixo f ou F permite a criação de strings formatadas. Strings formatadas permitem a execução de código Python cujo resultado será colocado no lugar do código. Por exemplo: `a = f'abs: {abs(-3)}'` é o mesmo que `a = 'abs: 3'`. O código `abs(-3)` será executado e substituído. Variáveis também podem ser utilizadas.

Há 3 notações especiais para conversores: "!s" para a função **str**, "!a" para a função **ascii** e "!r" para a função **repr**.

```
1 nome = 'João'
2 f'Ele disse se chamar {nome!r}' # Ele disse se chamar 'João'
3 f'Ele disse se chamar {nome!s}' # Ele disse se chamar João
4 f'{nome:10}' # 'João      '
```

Toda string formatada é passada pela função **format**. Há muitas opções de formatações, tanto que o documento de especificações onde as opções são listadas o fazem numa sessão entitulada "Format Specification Mini-Language". Você pode ver o documento de especificação na url `https://docs.python.org/3.6/library/string.html#formatspec` (disponível apenas em inglês).

Um exemplo simples, para demonstrar a mini-linguagem:

```
nome = 'João'
f'{nome:<10}' # 'João      '
f'{nome:>10}' # '      João'
f'{nome:^10}' # '   João   '
```

A Tabela 3 mostra os valores de escape presentes em Python.

Tabela 3 – Sequências de Escape

Sequência	**Significado**
\newline	Barra invertida e nova linha ignorados
\\	Barra invertida (\)
\’	Aspas simples (’)
\”	Aspas duplas (”)
\a	ASCII Sino (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Nova Linha (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Tabulação Horizontal (TAB)
\v	ASCII Tabulação Vertical (VT)
\ooo [1]	Caractére com valor octal ooo
\xhh [2]	Caractére com valor hexadecimal hh
\N{name}	Caractére Unicode chamado *name*
\uxxxx [3]	Caractére Unicode com valor hex 16-bit *xxxx*
\Uxxxxxxxx [4]	Caractére Unicode com valor hex 32-bit *xxxxxxx*

Sequências não reconhecidas eram ignoradas e tratadas como texto até a versão 3.6. Nesta versão um aviso **DeprecationWarning** é gerado. Versões futuras apresentarão o erro **SyntaxError**.

3.2.2 Concatenação de Strings

Strings podem ser concatenadas usando o operador +. Também é possível simplesmente escrever as duas strings uma após a outra ou em várias linhas usando parênteses.

```
a = 'Hello' + 'World'
b = 'Hello' 'World'
c = ('Hello'
'World')
```

[3] Até 3 dígitos octais.
[4] Obrigatório 2 digitos hexadecimais.
[5] Obrigatório 4 digitos hexadecimais.
[6] Obrigatório 8 digitos hexadecimais. Qualquer valor Unicode pode ser representado desta maneira.

3.2.3 Métodos Comuns

Strings em Python se comportam como sequências e podem ser indexadas.

```
msg = 'Hello World!'
msg[0] # 'H'
msg[-1] # '!'
msg[0:5] # 'Hello'
msg[6:] # 'World!', para iniciar do 0 ou ir até o final, o número pode ser omitido
msg[3:-3] # 'lo Wor'
```

Alguns métodos mais utilizados são listados abaixo.

```
len('Curso de Python') # 15
'Curso de Python'.lower() # 'curso de python'
'Curso de Python'.upper() # 'CURSO DE PYTHON'
'Curso de Python'.split(' ') # ['Curso', 'de', 'Python']
'Curso de Python'.find('de') # 6
'Curso de C++'.replace('C++', 'Python') # 'Curso de Python'
```

3.3 Lista

Lista é um tipo de sequência, onde cada elemento é associado a um índice. O primeiro índice é o 0. Elementos podem ser acessados usando colchetes. Os elementos da lista não precisam ser do mesmo tipo e a lista pode mudar de tamanho.

A maior parte das funcionalidades listadas aqui são funcionalidades de objetos que implementam a interface **sequence**, e não são exclusivos de listas. Strings, por exemplo, são sequências e podem ser utilizadas no lugar da lista na maioria dos exemplos mostrados aqui.

A interface **collection** é parecida com a interface **sequence**, porém com diferenças de implementação, pois sequências são ordenadas e coleções não. Ao iterar sobre uma coleção, não se tem garantia de ordem. Teoricamente é possível que ao iterar duas vezes sobre a mesma sequência se obtenha os valores em ordens diferentes, embora isso normalmente não aconteça.

```
ns = ['a', 'b', 'c', 'd', 'e', 'f']

len(ns) # tamanho da lista, 6 neste caso
```

```
4
5  n = ns[0] # n = 'a'
6  n = ns[-1] # n = 'f'
7
8  ns[5] = [15, 16, 17] # ns = ['a', 'b', 'c', 'd', 'e', [15, 16, 17]]
9  n = ns[5][1] # n = 16
10
11 # slicing
12 n = ns[2:5] # n = ['c', 'd', 'e'], que são os elementos 2, 3 e 4
13 n = ns[:3] # n = ['a', 'b', 'c']
```

3.3.1 Manipulação de Elementos

Para acessar um elemento utiliza-se colchetes. Os índices podem ser inteiros positivos ou negativos. Caso um objeto seja passado que não é um inteiro um erro é gerado (**TypeError**). Se o índice for maior que o último índice, a excessão **IndexError** é gerada.

O índice positivo vai até `len(lista)-1`. O índice negativo vai de -1 até `-len(lista)` e acessa a lista de trás pra frente. Listas aninhadas são acessadas repetindo o operador e passando os índices da lista mais exterior para a mais interior.

```
1  ns = ['a', 'b', 'c', 'f', 'd', 'e']
2
3  del ns[3] # ns = ['a', 'b', 'c', 'd', 'e']
4  ns.append('f') # ns = ['a', 'b', 'c', 'd', 'e', 'f']
5  ns.count('b') # 1, o número de vezes que 'b' aparece na lista
6  ns.extend([1,2,3]) # ns = ['a', 'b', 'c', 'd', 'e', 'f', 1, 2, 3]
7  ns.index(1) # 6, menor índice em que 1 aparece
8  ns.pop() # remove e retorna último elemento da lista
9  ns.pop(7) # remove e retorna 7º elemento da lista (o número 2)
10 ns.remove(1) # remove o elemento 1
11 ns.reverse() # Reverte a lista (ver também reversed(ns))
12 ns.sort() # Ordena a lista do menor pro maior (ver também sorted(ns))
13 ns.insert(0, 1) # insere o elemento 1 antes do elemento de índice 0
14 ns.clear() # limpa a lista
15 [1, 2, 3] + [4, 5, 6] # [1, 2, 3, 4, 5, 6]
16 ['Hi!'] * 4 # ['Hi!', 'Hi!', 'Hi!', 'Hi!']
17 3 in [1, 2, 3] # True se a lista tiver um elemento 3
18 max([1, 2, 3]) # 3
19 min([1, 2, 3]]) # 1
20 xs = [[1, 2, [3, 4, 5]], [6, 7, 8]]
21 xs[0][2][1] # 4
```

Outra forma de indexar sequências é utilizando *slices*. *Slices* permitem a criação de uma sublista de outra lista. A lista gerada é um cópia de parte a lista original. O início e final da sublista são separados por ponto-e-vírgula, sendo o primeiro índice inclusivo e o segundo exclusivo.

```
1 ns = list(range(10)) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2 xs = ns[2:-2]        # [2, 3, 4, 5, 6, 7]
```

3.3.2 Stacks

Stack (também conhecido como pilha) é uma forma de gerenciar dados no modelo *last-in, first-out*, onde o último elemento a ser inserido no *stack* será o primeiro a ser removido. Os métodos **append** e **pop** podem ser usados para gerenciar o stack numa lista de maneira eficiente.

Stacks são utilizados, por exemplo, para gerenciar a recursão (feito pelo interpretador), gerenciar uma fila de "fazer/desfazer" ou para reverter uma lista ou tabela.

```
1 ns = [1, 2, 3]
2
3 ns.append(4) # ns = [1, 2, 3, 4]
4 ns.append(5) # ns = [1, 2, 3, 4, 5]
5 ns.pop() # ns = [1, 2, 3, 4]
6 ns.append(6) # ns = [1, 2, 3, 4, 6]
7 ns.pop() # ns = [1, 2, 3, 4]
8 ns.pop() # ns = [1, 2, 3]
9 ns.pop() # ns = [1, 2]
```

3.3.3 Queues

Queues (filas) seguem o modelo *first-in, first-out*, onde o primeiro elemento inserido é o primeiro a ser removido. A classe lista não foi feita para trabalhar desta forma, portanto usa-se a classe **deque**, que é uma lista otimizada para inserção/remoção tanto no incío quanto no fim.

```
1 from collections import deque
2
3 ns = deque([1, 2, 3])
4
```

```
ns.append(4) # ns = [1, 2, 3, 4]
ns.append(5) # ns = [1, 2, 3, 4, 5]
ns.popleft() # ns = [2, 3, 4, 5]
ns.popleft() # ns = [3, 4, 5]
```

3.3.4 List Comprehension

List Comprehension é uma forma compacta de gerar listas a partir de outras listas. Compare os dois blocos de código a seguir, que geram uma lista dos quadrados de 0 até 9 ([0, 1, 4, 9, 16, 25, 36, 49, 64, 81]):

```
# Usando for explicitamente
xs = []
for x in range(10):
    xs.append(x**2)

# Usando List Comprehension
xs = [x**2 for x in range(10)]
```

Também é possível fazer o equivalente a vários for aninhados:

```
# Usando for explicitamente
xs = []
for x in range(10):
    for y in range(10):
        xs.append([x,y])

# Usando List Comprehension
xs = [[x,y] for x in range(10) for y in range(10)]
```

E usar condicionais:

```
# Usando for explicitamente
xs = []
for x in range(10):
    for y in range(10):
        if x != y:
            xs.append([x,y])

# Usando List Comprehension
xs = [[x,y] for x in range(10) for y in range(10) if x != y]
```

3.4 Tuples

Tuple é como uma lista, mas não pode ser modificada. Por esse motivo é também uma estrutura mais leve e simples.

Tuples são bastante utilizados em Python para agrupar valores, parecido com o *struct* do C/C++. Por ser muito utilizado a sintaxe permite sua criação e uso de forma simples.

A sintaxe é parecida com a de listas, mas usando parênteses no lugar de colchetes. Também é possível, em algumas situações, ignorar os parênteses, como na criação de variáveis ou na instrução **return**.

Duas outras notações interessantes são a de *pack* e *unpack*. Com estas notações variáveis podem ser agrupadas ou desagrupadas. *Packing* é simplesmente a criação de um *tuple* sem parênteses.

Unpacking é a separação de um *tuple* (ou qualquer sequência) em várias variáveis. Cada variável recebe um valor do *tuple*. É necessário que todos os valores sejam colocados em variáveis (mesmo número de variáveis e valores).

A notação `*var`, normalmente utilizada na declaração de funções, pode ser usada para criar uma lista com todos os elementos restantes, permitindo que haja menos variáveis que elementos na sequência.

```
1  xs = 1, 2, 3 # packing, o mesmo que xs = (1, 2, 3)
2
3  def random():
4      return 1, 2, 3
5
6  r = random() # r = (1, 2, 3)
7
8  a, b, c = xs # unpacking, a=1, b=2 e c=3 (também funciona com listas)
9  a, *b = xs # a=1, b=[2,3]
10 a, *b, c = list(range(10)) # a=0, b=[1, 2, 3, 4, 5, 6, 7, 8], c=9
```

3.5 Sets

Set é uma coleção sem elementos duplicados. Por isso é bastante utilizada para remover elementos duplicados de listas já existentes (desde que a ordem não importe). A verificação de existência de elemento é otimizada em *sets*.

Sets também podem ser vistos como conjuntos matemáticos. As operações que se aplicam a conjuntos matemáticos também podem ser utilizados em *sets*. A Tabela 4 mostra os operadores Python equivalentes a operações matemáticas.

Tabela 4 – Operações Matemáticas em *Sets*

Operador Matemático	**Operador Python**	**Exemplo**
$\bigcup$ (União)	|	`a | b`
$\bigcap$ (Interseção)	&	`a & b`
Diferença	−	`a - b`
Diferença Simétrica	∧	`a ^ b`

Outros métodos interessantes são **isdisjoint** que retorna **True** se não houver interseção, **issubset** que retorna **True** se este *set* conter o outro e **issuperset** que retorna **True** se este *set* for contido no outro.

```
1  xs = {1, 2, 3}
2  1 in xs # True, implementação mais eficiente
3
4  a = set('abracadabra') # a = {'a', 'r', 'b', 'c', 'd'}
5  b = set('alacazam') # b = {'c', 'l', 'm', 'z', 'a'}
6  c = a - b # c = {'r', 'b', 'd'}, letras em a mas não em b
7  c = a | b # c = {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'} letras em a ou b ou ambos
8  c = a & b # c = {'a', 'c'}, letras tanto em a quanto em b
9  c = a ^ b # c = {'l', 'm', 'b', 'z', 'r', 'd'}, letras em a ou b, mas não em ambos
10
11 # Set Comprehension
12 a = {x for x in 'abracadabra' if x not in 'abc'} # a = {'r', 'd'}
```

3.6 Comparação de Sequências

Sequências são comparadas item a item. Se um item for diferente, a comparação para e o resultado é o resultado da comparação deste item. Se uma sequência for um *slice* do início da outra, a sequência mais curta é considerada menor. Se os items comparados forem sequências do mesmo tipo, eles são comparados de forma recursiva. Comparação de strings utiliza os valores dos *code point Unicode* para decidir quem é maior.

```
1  # Todos retornam True:
2  (1, 2, 3)              < (1, 2, 4)
3  [1, 2, 3]              < [1, 2, 4]
```

```
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

3.7 Dicionários

Dicionário é uma coleção onde o indíce é um objeto dado. A maneira mais fácil de entender dicionários é pensar neles como pares de chaves e valor, onde você guarda um valor em determinada chave, que será única no dicionário.

Para ser chave de um dicionário o objeto deve ser imutável. Números e *strings* são comumente utilizados. *Tuples* também podem ser usados como chaves, desde que todos os elementos sejam imutáveis.

Internamente o que é guardado como chave não é o objeto em si, mas um *hash* do objeto, por isso não há diferença na busca por conta de complexidade do objeto.

Dicionários também são declarados utilizando chaves, como *sets*, no entanto os items são pares de chave/valor separados por dois-pontos. Há um conflito de sintaxe quanto ao dicionário/set vazio, já que ambos seriam criados por `{}`. Esta sintaxe cria um dicionário vazio, para criar um *set* vazio utilize o construtor: `set()`.

Assim como em listas, também é possível utilizar colchetes para acessar e modificar valores no dicionário, no entanto o índice é qualquer objeto imutável. Por ser desordenado, novos elementos podem ser adicionados em qualquer "posição". A posição do item no dicionário não é assegurado.

```
tel = {'joão': 4098, 'josé': 4139}
tel['ana'] = 4127 # tel = {'joão': 4098, 'josé': 4139, 'ana': 4127}
tel['joão'] # 4098
del tel['josé'] # tel = {'joão': 4098, 'ana': 4127}
list(tel.keys()) # or list(tel), ['joão', 'ana']
sorted(tel.keys()) # or sorted(tel), ['ana', 'joão']
'joão' in tel # True
'joão' not in tel # False
dict([('joão', 4139), ('josé', 4127), ('ana', 4098)])
dict(joão=4139, josé=4127, ana=4098)
```

A estrutura do *Dictionary Comprehension* também é parecida com a do *Set Comprehension*, se diferenciando pela presença dos dois-pontos.

```
# Dictionary Comprehension
{x: x**2 for x in range(10)}
```

Para iterar sobre um dicionário pode-se iterar sobre suas chaves, valores ou ambos ao mesmo tempo.

```
tel = {'jack': 4098, 'sape': 4139, 'guido': 4127}

# Iterando sobre as chaves:
for key in tel.keys():
    print(tel[key])

# Iterando sobre os valores:
for value in tel.values():
    print(value)

# Iterando sobre ambos:
for k,v in tel.items():
    print('%s: %s' % (k,v))
```

Capítulo 4

Controle de Fluxo

4.1 If

Condicionais permitem a execução de código apenas se uma certa expressão for verdadeira. Em Python essa execução condicional é feita com a estrutura **if**. O resultado da expressão condicional não precisa ser um valor booleano, ele pode ser uma expressão que seja considerada verdadeira ou falsa naquela condição. Por exemplo `4`, `'ha'` e `[1,2]` são considerados verdadeiros enquanto `0`, `''` e `[]` são considerados falsos.

Existem duas formas de se utilizar o **if** em Python:

```
1  # Método 1:
2  if a == 2 or b != 3:
3      do_something()
4  elif 1 <= c < 5:
5      do_something_else()
6  else:
7      do_nothing()
8
9  # Método 2, equivalente ao operador ternário do C (a ? b : c)
10 a = 1 if b else 2
```

Diferente de linguagens como C, os operadores booleanos não retornam um valor booleano em Python, mas sim o último valor passado. Por isso é possível criar expressões condicionais que resultam em um valor sem utilizar a estrutura **if**. Essas expressões podem ser utilizadas para definir valor padrão para variáveis usando uma sintaxe mais curta. Blocos **if/else** simples também podem ser substituídos.

```python
# Uma forma de definir valor padrão (caso a variável seja None)
album = getAlbum() or 'Sem Album' # se getAlbum() retornar None, o valor
                                  # 'Sem Album' será atribuído à variável

a = 2
b = a % 2 == 0 and 'sim' or 'não' # como 2 é múltiplo de 2, b = 'sim'
c = a % 3 == 0 and 'sim' or 'não' # como 2 não é múltiplo de 3, c = 'não'

# A função range muda o significado dos parâmetros dependendo do número de
# parâmetros informados. Eis um exemplo de implementação.
# assinaturas:
# range(stop)
# range(start, stop)
# range(start, stop, step)
def range(start, stop=None, step=1):
    start, stop = stop is None and (0, start) or (start, stop)
    xs = []
    while start < stop:
        xs.append(start)
        start += step
    return xs
```

4.2 For

O loop **for** em Python é utilizado para iterar uma sequência. Sequência é uma interface e qualquer objeto que implemente um determinado conjunto de métodos é considerado uma sequência. Os exemplos mais comuns da biliboteca padrão são **list**, **set**, **tuple**, **dictionary** e **string**.

```python
for a in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print(a)
```

Podemos criar uma lista de n até m-1, com passo x utilizando a função **range**. Se x for omitido ele será 1. Se apenas um valor for informado, este valor será m e n será 0.

```python
for a in range(n, m, x):
    print(a)

for a in range(10): # o mesmo que range(0, 10, 1)
    print(a)
```

Loops podem ter sua execução controlada pelas instruções **continue** e **break**. A instrução **continue** faz com que a execução do corpo pare mas os próximos items sejam executados normalmente. A instrução **break** para a execução do loop.

```
for a in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    if a == 5:
        continue # O item 5 não será escrito, mas 6, 7, 8 e 9 serão.
        print(a)

for a in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    if a == 5:
        break # O loop para ao chegar no item 5, sendo escritos apenas os items 0, 1, 2, 3 e 4.
        print(a)
```

4.3 For...else

Em Python é possível específicar um bloco **else** para a estrutura **for**. Este bloco só será executado se a lista for percorrida até o final, ou seja, se não ocorrer um **break**.

```
for x in [1, 3, 5, 7, 9, 10, 11, 13]:
    if x % 2 == 0:
        print('Tem um número par na lista!')
        break
else:
    print('Não há número par na lista!')
```

4.4 While

O loop **while** em python é bem parecido com o de outras linguagens. Não há **do...while** em Python.

```
while condition:
    do_something()
```

As palavras-chave **break** e **continue** podem ser usadas tanto com loops **for** quanto **while**.

```
a = 0
while a < 10:
    if a == 5: # pula o número 5
        continue
    elif a > 7: # para o loop no número 7
        break
    print(a)
    a += 1
```

4.5 While...else

Um bloco **else** também pode ser usado com o **while**.

```
xs = [1, 3, 5, 7, 9, 10, 11, 13]
i = 0
while i < len(xs):
    if xs[i] % 2 == 0:
        print('Há um número par na lista!')
        break
    i += 1
else:
    print('Não há número par na lista!')
```

4.6 Pass

Como Python usa indentação para definir escopo, não é possível ter um escopo vazio. Mas as vezes é necessário definir um escopo vazio para algo que será implementado mais tarde ou em outro lugar (herança, metaprogramação). Para isso utiliza-se a instrução **pass**, que não faz nada.

```
if False:
    pass

for x in range(10):
    pass

while False:
    pass
```

4.7 Switch

Não há uma expressão **switch** no Python, no entanto o mesmo resultado pode ser obtido usando um dicionário.

```
1  def str_to_int(h):
2      return { '0': 0,
3               '1': 1,
4               '2': 2,
5               '3': 3,
6               '4': 4,
7               '5': 5,
8               '6': 6,
9               '7': 7,
10              '8': 8,
11              '9': 9 }.get(h, None)
```

Capítulo 5

Funções

Funções em Python são definidas usando a seguinte sintaxe:

```
1 def function(arg1, arg2=default, *varg, **kwarg):
2     """docstring"""
3     do_something()
4     return something, something_else
```

onde:

- *arg1* é um parâmetro obrigatório. A posição na chamada define seu valor.
- *arg2* é um parâmetro optativo. Se não for passado, terá o valor de default
- *varg* é um parâmetro variável. Permite-se apenas um na função. Trata-se de um vetor com todos valores passados após o último argumento posicional.
- *kwarg* é um parâmetro nomeado variável. Qualquer argumento nomeado que for passado mas não estiver definido na assinatura da função será colocado neste dicionário. Permite-se apenas um na função.

Docstring é uma string que documenta a função. Ela também pode estar presente em classes e módulos. No terminal interativo pode-se usar a função **help** para ler esta documentação (Exemplo: `help(range)`). Ela também é utilizada por algumas ferramentas para extrair a documentação e criar um manual externo.

5.1 First Class Citizens e High Order

First Class Citizens significa que funções são objetos como outro qualquer. Podem ser guardadas em variáveis, referenciadas, e, de certa forma, manipuladas. *High Order* significa que funções podem receber funções como argumentos e retornar funções.

As funções em Python são *First Class Citizens* e *High Order* pois todo valor na linguagem é um objeto, inclusive funções, classes e módulos. Este objetos tem, inclusive, métodos e atributos, como o atributo **___doc___**, que retorna a *docstring* do objeto.

Não é possível fazer *overload* em Python, mas isto normalmente não é necessário devido à natureza dinâmica da linguagem. Utilizando uma combinação de parâmetros obrigatórios, opcionais, variáveis e arbitrários pode-se conseguir o mesmo efeito. Na verdade, parâmetros arbitrários aumentam muito as possibidades e o dinâmismo da linguagem permite todas as permutações escrevendo apenas um método.

```
1  def sum(*cs):
2      s = 0
3      for c in cs:
4          s += c
5      return s
6
7  def apply(function, arguments):
8      return function(*arguments)
9
10 def apply_one(function):
11     return lambda *cs: function(1, *cs)
12
13 r = apply(apply_one(sum), [2, 3, 4, 5]) # equivalente a sum(1, 2, 3, 4, 5)
```

5.2 Parâmetros

Parâmetros são os valores que uma função espera receber, as suas variáveis de entrada. Argumentos são os valores passados para a função na chamada. No entanto é comum ver os termos parâmetro e argumento serem utilizados indiscriminadamente.

Python aceita parâmetros posicionais, que são aqueles em que os parâmetros recebem seus valores respeitando a ordem de declaração dos parâmetros e a ordem de passagem dos argumentos.

Em Python os argumentos também podem ser passados nomeadamente. Para isso basta atribuir o valor desejado ao parâmetro, como se o nome do parâmetro fosse uma variável. Desde que todos os parâmetros sejam dados, uma vez que nomeados, a ordem não importa.

5.2.1 Parâmetros Obrigatórios

Parâmetros obrigatórios terão seu valor definido pela sua posição na declaração da função. Se forem passados nomeadamente, a posição deixa de importar.

```
1  def subtract(a, b):
2      return a - b
3
4  subtract(1, 2) # fará 1 - 2, sendo a=1 e b=2
5  subtract(2, 1) # fará 2 - 1, sendo a=2 e b=1
6
7  subtract(a=1, b=2)
8  subtract(b=2, a=1) # Igual a chamada acima.
```

5.2.2 Parâmetros Opcionais

Parâmetro opcional é um parâmetro que possui um valor padrão. Se ele não for informado pelo usuário, seu valor padrão será utilizado.

```
1  def subtract(a, b=1):
2      return a - b
3
4  subtract(1) # fará 1 - 1, sendo a=1 e b=1 (o valor padrão)
5  subtract(2, 5) # fará 2 - 5, sendo a=2 e b=5
```

Parâmetros opcionais sempre devem ser declarados após os parâmetros obrigatórios. Estes também podem ser nomeados, mas não pode haver parâmetro posicional após parâmetro nomeado.

```
1  subtract(a=2, b=1)
2  subtract(2, b=1) # Igual a chamada acima.
```

5.2.3 Parâmetros Variáveis

Parâmetros variáveis são utilizados para aumentar a quantidade de argumentos recebidos pela função indefinidamente. Ele deve ser posicionado após os argumentos obrigatórios e opcionais. Apenas um é permitido na função, e ele será uma lista contendo todos os valores passados após o último argumento obrigatório ou opcional.

```
1  def produtorio(a, b, *cs):
2      r = a * b
3      for c in cs:
4          r *= c
5      return r
6
7  produtorio(1, 2, 3, 4, 5, 6, 7, 8, 9) # a=1, b=2, cs=[3, 4, 5, 6, 7, 8, 9]
```

A notação com * também pode ser utilizada na chamada de funções para transformar uma lista em argumentos posicionais. Isto é útil para construir a lista de argumentos dinâmicamente (por exemplo, com dados inseridos pelo usuário ou tirados de um banco de dados) e chamar uma função com estes argumentos.

```
1  vs = list((range(1, 10)) # [1, 2, 3, 4, 5, 6, 7, 8, 9]
2  produtorio(*vs)
```

5.2.4 Parâmetros Arbitrários

Funcionam como os parâmetros variáveis, mas devem ser nomeados. Devem ser a última coisa na lista de parâmetros. A variável é um dicionário com os pares de chave/valor que forem passados.

```
1  def table_print(**kwarg):
2      for k,v in kwarg.items():
3          print('%s\t%s' % (k,v))
4
5  table_print(a=1, b=2, c=3)
6
7  ds = {'a': 1, 'b': 2, 'c': 3}
8  table_print(**ds) # equivalente a table_print(a=1, b=2, c=3)
```

5.3 Lambdas

Lambdas são funções anônimas e curtas. Elas podem receber qualquer número de parâmetros e são uma única expressão. Elas acessam apenas as variáveis em sua lista de parâmetros e no escopo global. Geralmente são expressões que transformam um valor.

```
1 add = lambda a, b: a + b
2 r = add(1, 2) # r = 3
3
4 import functools
5 sum = lambda *cs: functools.reduce(add, cs, 0)
6 r = sum(1, 2, 3) # r = 6
```

5.4 Retorno

Funções em Python sempre retornam um valor. No caso de funções que não tem uma instrução **return**, ou que tem esta instrução vazia, **None** é retornado.

```
1  def return_none(): # returns None
2      do_something()
3
4  def return_none_too(): # returns None
5      return
6
7  def return_one_value(): # returns 1
8      return 1
9
10 def return_multiple_values(): # returns (1, 2, 3)
11     return 1, 2, 3
```

5.5 Módulos

Módulo é um arquivo Python contendo código executável ou definições. O nome do módulo é o nome do arquivo sem a extensão ".py". Módulos são usados para quebrar o código em pequenas partes que atuam em uma mesma área do aplicativo. Por exemplo, tudo dentro do módulo padrão **os.path** tem a ver com caminhos no sistema de arquivo.

Se criarmos um arquivo "mat.py" contendo o seguinte código:

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

Poderemos importar este arquivo e usar esta função no terminal ou em outro arquivo:

```
1 import mat
2
3 print(mat.soma(1, 2))
```

É importante notar que módulos só são carregados (e executados) na primeira vez que são importados. Isso faz com que uma instrução **import** dentro de uma função que pode ser executada várias vezes não seja um problema, já que o interpretador não fará nada além de criar uma variável local apontando para o objeto do módulo.

O interpretador irá procurar por arquivos de módulo no "*search path*", que pode ser visto e manipulado através da variável **path** no módulo **sys**. A função **dir** lista todos os nomes declarados dentro de um módulo.

5.6 Pacotes

Módulos podem ser organizados em pacotes para melhor organizar o programa. Pacotes são diretórios contendo um arquivo chamado "__init__.py". Este arquivo pode estar vazio, mas normalmente contém o código de inicialização do pacote. Para importar um módulo de um pacote, usa-se um ponto para separar os nomes.

```
1 import os.path
2 import numpy.linalg
```

Exemplo de um arquivo "__init__.py":

```
1 __author__ = 'Álan Crístoffer'
2 __copyright__ = 'Copyright 2016, Álan Crístoffer'
3 __credits__ = ['Álan Crístoffer']
4 __license__ = 'MIT'
5 __version__ = '1.0.2'
6 __maintainer__ = 'Álan Crístoffer'
7 __email__ = 'acristoffers@gmail.com'
8 __status__ = 'Release'
```

Capítulo 6

Programação Orientada a Objetos

Python foi criada para ser uma linguagem orientada a objetos. Neste paradigma objetos são uma coleção de funções e variáveis. Funções que pertencem a um objeto são chamadas de métodos e as variáveis de atributos.

Como já foi dito, em Python tudo é objeto, mesmo funções, métodos, classes, módulos e pacotes. Por ter sido feita com a orientação a objetos em mente, a sintaxe para criação de classes e objetos é simples.

Classes são criadas em Python usando a palavra chave **class**. Variáveis de classe são definidas diretamente no escopo da classe, enquanto variáveis de instância são definidas no construtor. O construtor em Python é a função **___init___(self)**.

Para instanciar uma classe basta chamar a mesma, como se ela fosse uma função. De fato é isso que você está fazendo quando converte valores: criando novos objetos chamando a classe e passando como argumentos os valores a serem convertidos.

```
1 class Music:
2     """Esta classe define uma música"""
3     type = 'music' # variável de classe
4
5     def __init__(self): # construtor
6         self.genre = 'pop' # variável de instância.
```

Como pode ser observado no código acima, classes também suportam *docstrings*. A variável *type* é uma variável de classe. Para acessá-la usa-se o nome da classe: `Music.type`. Diferente de outras linguagens não se pode acessar uma variável de classe usando uma instância da mesma. A variável *genre* é uma veriável de instância, também conhecida como atributo. Atributos em Python são definidos diretamente no objeto **self**.

6.1 self

Métodos e funções em Python se diferem por um detalhe: métodos devem ter um primeiro parâmetro que não será passado pelo usuário, mas será provido pelo interpretador. Esse parâmetro referenciará o próprio objeto e por convenção é chamado de **self**. Você é fortemente encorajado a usar o nome **self** também. Ele é o equivalente ao **this** em linguagens como C++, Java e C#.

Assim como operadores são "convertidos" em chamadas de métodos no objetos envolvidos, chamar um método em um objeto é também "convertido" em uma chamada de método na classe que recebe como parâmetro o objeto que possui o método e os argumentos passados. Ambos entre aspas pois nenhum dos dois ocorrem explicitamente, mas os efeitos são os mesmos que se eles ocorrecem. Isso explica o porquê de se ter o primeiro parâmetro **self**.

```
1 class MinhaClasse():
2     def meu_metodo(self, parâmetro):
3         pass
4
5 meu_objeto = MinhaClasse()
6 meu_objeto.meu_metodo(1) # será convertido pelo interpretador em
7                          # MinhaClasse.meu_metodo(meu_objeto, 1)
```

6.2 Métodos

Métodos são funções que pertencem à classe ou ao objeto. O método pertencente ao objeto deve ter o parâmetro **self** enquanto o método de classe não. Algumas linguagens permitem chamar um método de classe em um objeto. Em Python isso não é possível.

```
1 class MinhaClasse():
2     def metodo_do_objeto(self):
3         pass
4     def metodo_da_classe():
5         pass
6
7 MinhaClasse.metodo_da_classe()
8 obj = MinhaClasse()
9 obj.metodo_do_objeto()
```

Python não tem palavras chaves para permissão de acesso, como *public*, *protected* e *private*. Na verdade, a linguagem nem reforça essas permissões. Em Python utiliza-se convenção para declarar a accessibilidade de um método, função ou atributo. Você pode acessar métodos *protected* de qualquer lugar e os *private* estão apenas escondidos, mas são acessíveis se você realmente quiser.

A lógica por trás desta escolha é que a liberdade deve ser do usuário de usar métodos protegidos e privados caso ele queira. Entende-se que o usuário da biblioteca sabe dos riscos ao fazê-lo não cabendo à linguagem proibí-lo.

```
1 class MinhaClasse():
2     def metodo_publico(self): # Todo método é normalmente público
3         pass
4 
5     def _metodo_protected(): # Métodos protegidos começam com um _
6         pass
7 
8     def __metodo_private(): # Métodos privados começam com 2 _
9         pass
```

6.3 Herança

Herança permite o reuso de código e facilita a manutenção, assim como o uso de funções. Python permite herança múltipla, o que significa que você pode extender mais de uma classe de uma vez, herdando funcionalidades de todas elas.

```
1  class A:
2      def __init__(self):
3          self.msg = 'hello'
4 
5  class B:
6      def hello(self):
7          print(self.msg)
8 
9  class C(A, B):
10     def __init__(self):
11         A.__init__(self)
12 
13 c = C()
14 c.hello()
```

Pode-se verificar se um objeto é uma instância de uma classe com o método **isinstance**. Também é possível verificar se uma classe é subclasse de outra usando o método **issubclass**. Toda classe em Python herda da classe **object**.

Quando você chama uma função ou acessa um atributo em um objeto o interpretador procura pelo nome primeiro na classe do objeto, depois nas classes herdadas, na mesma ordem em que foram definidas, e por último na classe **object**. O método de classe **mro** (*method resolution order*) retorna uma lista com as classes na ordem buscada.

6.4 Duck Typing

Em linguagens estaticamente tipadas o tipo do objeto diz se ele possui ou não determinada funcionalidade, o que pode ser verificado durante a compilação. Isso faz com que somente objetos que herdem de determinada classe ou implementem determinada interface possam ser usados em determinadas situações.

Em linguagens dinâmicas o tipo da variável não é conhecido a priori, o que impossibilita a ação do compilador. Um efeito parecido pode ser conseguido testando-se o tipo da variável durante a execução.

Mas se podemos testar o tipo da variável durante a execução, não podemos testar simplesmente se o objeto possui as funcionalidades que precisamos? A idéia é simples: se anda como um pato e grasna como um pato, deve ser um pato.

Duck Typing permite então que qualquer objeto que implemente uma certa interface possa ser utilizado para determinada função. Por exemplo, qualquer objeto que tenha o método **___add___** pode ser utilizado com o operador + e qualquer objeto com o método **___getitem___** pode usar a syntaxe de indexação com [].

Lembrando novamente que não há interfaces em Python como em Java. Interfaces aqui são apenas definições sem contrato, não reforçadas pela linguagem. Se um objeto implementa o método **___add___** ele implementa a interface de adição, mesmo sem herdar ou definir isto em lugar nenhum. Basicamente o que chamo de interfaces em Python são definições de *Duck Typing*.

Duck Typing está enraizado em Python. Por exemplo, para fazer um objeto que possa ser iterado usando **for** ou *List Comprehension* basta implementar os métodos **___iter___** e **___next___**.

```
1  class A:
2      def __init__(self, v):
3          self.value = v
4
5      def __add__(self, other):
6          return self.value + other.value
7
8  a = A(1)
9  b = A(2)
10 c = a + b # o mesmo que a.__add__(b), que retorna 3
11
12 class Contador:
13     def __init__(self):
14         self.final = 10
15
16     def __iter__(self):
17         class iterator:
18             def __init__(self, o):
19                 self.o = o
20                 self.atual = 0
21
22             def __next__(self):
23                 if self.atual < self.o.final:
24                     self.atual += 1
25                     return self.atual - 1
26                 else:
27                     raise StopIteration
28
29         return iterator(self)
30
31 print([x for x in Contador()])
```

Capítulo 7

Programação Funcional

Em ciência da computação, programação funcional é um paradigma - um forma de construir as estruturas e os elementos de um programa - que trata a computação como a resolução de funções matemáticas e evita dados mutáveis e mudança de estados. É um paradigma de programação declarativo, onde a programação é feita com expressões ao invés de instruções.

Na programação funcional a saída de uma função depende unicamente dos valores de entrada. Chamar uma função duas vezes com os mesmos argumentos irá sempre produzir os mesmos resultados. Isso se diferencia da programação procedural, onde mudança de estados permitem que a função retorne resultados diferentes para os mesmos argumentos, mas um estado diferente do programa. Um exemplo é a função **___next___**, que irá retornar um objeto diferente em cada chamada.

Eliminando os efeitos colaterais causados pelas mudanças de estados é possível criar programas que são fáceis de entender e de predizer o resultado, o que é uma das motivações para o desenvolvimento da programação funcional. Isso também facilita o desenvolvimento de aplicativos com concorrência, já que não há mais o problema de *data race*, onde o mesmo espaço de memória é escrito e lido ao mesmo tempo.

Na programação funcional tudo é tratado como dado, mesmo as funções. Por isso é importante para a aplicação deste paradigma que a linguagem tenha funções como First Class Citizens e que elas sejam High Order Functions.

7.1 Geradores

Gerador é um objeto que implementa uma interface especial, que o permite retornar um item por vez. Por exemplo, a função **range** retorna um gerador, e não uma lista. Você pode obter uma lista instanciando **list** e passando o gerador como parâmetro. Uma função pode ser um gerador se for utilizada a palavra chave **yield**.

O principal motivo para usar-se geradores é economia de memória. Você pode criar um objeto usando a função **range** que se comporta como uma lista de 0 a 10000000000000 sem de fato colocar todos estes números na memória.

Fazendo geradores que iteram sobre geradores é possível tratar grandes quantidades de dados como se eles estivessem em uma lista, mas recuperando-os aos poucos. Isto torna possível trabalhar todos os dados em um banco de dados de forma funcional sem grandes impactos na memória.

É importante notar que uma vez que o gerador é completamente percorrido, ele se esgota. Tentar usa-lo duas vezes seguidas fará com que a segunda vez seja equivalente a uma lista vazia. Se for necessário reusar um gerador, ele deve ser copiado antes de ser usado ou transformado em uma lista, que ocupará mais espaço na memória.

```
1  xs = range(10) # gerador
2  ys = (x**2 for x in range(10)) # gerador no estilo List Comprehension
3
4  def prime_generator():
5      yield 2
6      ps = []
7      c = 1
8      def up_to_square(y, xs):
9          for x in xs:
10             if y <= x**0.5:
11                 yield y
12             else:
13                 return
14     while True:
15         c += 2
16         if not any((c % p == 0 for p in up_to_square(c, ps))):
17             ps.append(c)
18             yield c
19
20 import itertools
21 primes = list(itertools.islice(prime_generator(), 100000))
```

7.2 Map

A operação mais simples na programação funcional é a **map**. A função **map** recebe uma função e uma lista como parâmetros e retorna um gerador que aplica a função em cada um dos elementos da lista dada.

List Comprehension é uma forma de **map** (desde que não seja utilizado o **if**, porém a função nativa deve ser preferida se velocidade for um problema, já que a função **map** é implementada em código nativo (baixo nível) e a *List Comprehension* vira um *loop* **for** implementado em Python (alto nível).

```
1 xs = list(range(10))
2 ys = [x**2 for x in xs]
3 zs = map(lambda x: x**2, xs)
```

Para ter uma ideia da diferença de performance, compare estes resultados [1]:

%timeit [x**2 for x in range(100000)]
36.2 ms ± 929 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

%timeit map(lambda x: x**2, range(100000))
545 ns ± 8.27 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

7.3 Reduce

A operação **reduce** transforma uma lista em um único elemento através da aplicação de uma função em pares de elementos da lista. Em Python 3 esta função foi transferida para o módulo functools.

Um exemplo seria o operador matemático somatório. Dado uma lista de elementos, a operação soma será aplicado aos dois primeiros items, e o resultado será somado ao terceiro item, e este resultado somado ao quarto, etc:

$\sum x, x \in \{1, 2, 3, 4, 5\}$ se expande para (((1 + 2) + 3) + 4) + 5.

[1] Benchmark feito uma vez usando IPython

Também é possível passar um elemento inicial, que será usado na primeira operação junto com o primeiro elemento da lista. Este elemento também será retornado caso a lista esteja vazia. Um elemento neutro à operação pode ser utilizado para garantir o retorno de um valor, como o 0 para a soma.

```
1 from functools import reduce
2
3 def sum(xs):
4     return reduce(lambda a, b: a + b, xs)
5
6 def fib(n):
7     return reduce(lambda a, b: (a[1], sum(a)), range(n), (0, 1))[1]
```

7.4 Filter

A função **filter** filtra retorna um gerador cujos items serão apenas aqueles para quais a função dada retornar um valor verdadeiro. É como usar uma *List Comprehension* onde o valor é retornado inalterado mas com a presença de uma cláusula **if**. Se **None** for passado no lugar da função, apenas os valores verdadeiros serão retornados.

```
1 def is_prime(n):
2     return not(n<2 or n>2 and (n%2==0 or any((n%x==0 for x in range(3, int(n**0.5)+1, 2)))))
3 primes = filter(is_prime, range(100))
4
5 str = 'x714n19y8dufhvosd`c8798\$#@!\%!5vsdhjasf'
6 only_alpha = filter(lambda c: c.isalpha(), str) # equivalente: (c for c in str if c.isalpha())
7 ''.join(only_alpha) # 'xnydufhvosdcvsdhjasf'
```

7.5 Sorted

A função **sorted** retorna uma lista com os items da lista de entrada organizados por ordem alfabética. O parâmetro *key* é uma função que será aplicada em cada elemento antes da comparação. A comparação será feita entre os elementos retornados pela função passada. O parâmetro *reverse* pode ser usado para ordenar em ordem reversa (*default*: **False**).

```
1  str = 'abracadabra'
2  cs = sorted(str) # ['a', 'a', 'a', 'a', 'a', 'b', 'b', 'c', 'd', 'r', 'r']
3
4  xs = [71, 66, 45, 94, 19, 25, 79, 54, 59, 23]
5  xs = sorted(xs) # [19, 23, 25, 45, 54, 59, 66, 71, 79, 94]
```

7.6 Zip

A função **zip** retorna um gerador onde cada item é um tuple contendo os n-ésimos items de várias listas. A lista retornada será tão grande quanto a menor lista, ou seja, o gerador se esgota quão logo o primeiro gerador se esgota.

```
1  xs = [1, 2, 3]
2  ys = [4, 5, 6]
3  zs = [7, 8, 9]
4
5  zip(xs, ys, zs) # gerador para a lista [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Capítulo 8

Matemática Computacional

A matemática computacional explora o poder de processamento dos computadores para realizar cálculos e resolver problemas que seriam difíceis ou inconvenientes de serem resolvidos à mão.

Os softwares mais conhecidos de matemática computacional são o *MATLAB* e o *Octave*. Essas linguagens são feitas para computação numérica, e fazem isso de forma simples. No entanto elas são péssimas para coisas mais genéricas que se fazem necessária, como criação de interface gráfica, manipulação textual e interfaceamento com programas. Até mesmo orientação a objeto foi feito nestas linguagens como um remendo.

Para resolver esses problemas Travis Oliphant, Eric Jones, and Pearu Peterson resolveram usar a linguagem Python, que é genérica, e escrever bibliotecas que facilitassem a computação numérica nesta linguagem. Nasceu assim a *toolkit SciPy*.

8.1 SciPy

SciPy é uma coleção de software open-source para computação científica escrita em Python, e particularmente um conjunto de pacotes.

- **NumPy** O pacote fundamental. Define vetores e matrizes e as operações matriciais elementares.

- **SciPy** Uma coleção de algorítmos numéricos e *toolboxes* especializados, incluindo processamento de sinais, otimização, estatística, entre outros.

- **Matplotlib** Pacote para geração de gráficos 2D com qualidade de publicação e 3D rudimentar.
- **Pandas** Estruturas de dados de alta performance e fácil uso.
- **SymPy** Matemática simbólica e álgebra computacional.
- **IPython** Interface interativa em forma de *notebooks*, similar ao Mathematica.

8.2 NumPy

Numpy é uma biblioteca de processamento numérico. O objeto principal da NumPy é o **array**. Ele é um vetor multidimensional homogênio. É como uma tabela de elementos, todos do mesmo tipo, indexados por um conjunto de inteiros positivos. Na NumPy, dimensões se chamam *axes* e o número de dimensões é o *rank*.

Por exemplo, se declararmos uma matriz para representar dois pontos no espaço, ele terá *rank* 2 (duas dimensões) e o tamanho do *axis* será 3:

```
1 import numpy as np
2
3 pontos = np.array({{[}}[1, 0, 0], [0, 1, 0]])
4
5 pontos.shape   # (2, 3) - forma
6 len(pontos)    # 2      - tamanho da primeira dimensão
7 len(pontos[0]) # 3      - tamanho da segunda dimensão
8 pontos.ndim    # 2      - número de dimensões
```

Há 613 nomes definidos no módulo numpy, abrangendo funções, variáveis e submódulos. Estas funções são tanto para criação e manipulação de arrays quanto funções matemáticas, como raiz quadrada, seno e cosseno, além de vários equivalentes a funções MATLAB, como *meshgrid*.

8.3 Arrays

Para criar um array basta instanciar a classe **array** passando um vetor Python como parâmetro, ou usar uma das funções especiais.

```
import numpy as np
import numpy.matlib as ml

definido = np.array([[1, 0], [0, 1]])
identidade = np.eye(3)
zeros = np.zeros( (4, 4) )
ums = np.ones( (5, 2) )
aleatorio = np.random.rand(3, 4)
repetida = ml.repmat(np.eye(2), 2, 2)
ajustada = np.arange(9).reshape(3, 3)
linearmente_espacado = np.linspace(0, 100, 1000, endpoint=True)
```

8.3.1 Operações Básicas

Os operadores normais $(+, -, *, /, **, \wedge, \%)$ são aplicados elemento-a-elemento. Um novo array é criado e o resultado guardado nele.

```
import numpy as np

a = np.array([20, 30, 40, 50])
b = np.array([0, 1, 2, 3])

a - b                          # [20, 29, 38, 47]
b ** 2                         # [0, 1, 4, 9]
a < 35                         # [True, True, False, False]
a % (b + 1)                    # [0, 0, 1, 2]
np.array([True, False]) | True # [True,  True]
np.sin(b)                      # [0, 0.84147098, 0.90929743, 0.14112001]
```

8.3.2 Operações Matriciais e In-Place

O produto matricial pode ser calculado usando o método **dot**. Na versão 5 do Python foi introduzido o operador @ que faz multiplicação matricial. Algumas operações são realizadas *in-place*, sem criar um novo array, como += e -=.

```
import numpy as np

a = np.arange(1, 10).reshape(3, 3)
b = a.T # transposta

a.dot(b)     # [[ 14,  32,  50],
np.dot(a, b) #  [ 32,  77, 122],
```

```
a @ b        #  [ 50, 122, 194]]

c = a.copy() # cria uma cópia do array
c *= b # [[ 14,  32,  50],
       #  [ 32,  77, 122],
       #  [ 50, 122, 194]]
```

8.3.3 Indexação

Arrays unidimensionais se comportam como listas e todos as operações de listas são suportadas. Arrays multidimensionais podem ter um índice pra cada dimensão. Os índices são separados por vírgula.

```
import numpy as np

a = np.arange(1, 10).reshape(3, 3)
a[2,2]   # 9
a[:,2]   # [3, 6, 9]
a[1:2,:] # [[4, 5, 6]]
a[2] == a[2, :]
a[1, ... ] # ...  será substituído por quantos dois pontos forem necessários
           # e pode estar presente no meio da expressão, como no unpacking

b = a.flatten() # equivalente ao m(:) do MATLAB
```

8.3.4 Slicing

Iterar sobre um array multidimensional é iterar sobre a primeira dimensão.

```
import numpy as np

a = np.arange(1, 10).reshape(3, 3)

for x in a:
    print(x) # [1, 2, 3], [4, 5, 6], [7, 8, 9]

for x in a.flat: # flat retorna um iterador ao invés de uma nova lista
    print(x) # 1, 2, 3, 4, 5, 6, 7, 8, 9
```

8.3.5 Manipulação de Forma

A forma do array pode ser modificada pelas funções **reshape**, **resize**, **ravel**, **flatten** e **T**.

```
import numpy as np

a = np.arange(1, 13).reshape(4, 3)

b = a.reshape(3, 4) # Altera a forma de (4,3) para (3,4).
                    # Preste atenção na ordem dos elementos.

a.resize((3,4)) # O mesmo que reshape, mas altera o objeto ao invés de criar
                # uma cópia alterada.

a.ravel() # o mesmo que a.flatten(), retorna todos os elementos em um vetor.

a.T # Transposta de a
```

8.3.6 Concatenação

Arrays podem ser concatenados usando a função **concatenate**

```
import numpy as np

a = np.arange(9).reshape(3, 3)
b = np.arange(9, 18).reshape(3, 3)

c = np.concatenate( (a, b) ) # [[ 0,  1,  2],
                             #  [ 3,  4,  5],
                             #  [ 6,  7,  8],
                             #  [ 9, 10, 11],
                             #  [12, 13, 14],
                             #  [15, 16, 17]]

d = np.concatenate( (a, b), axis=1 ) # [[ 0,  1,  2,  9, 10, 11],
                                     #  [ 3,  4,  5, 12, 13, 14],
                                     #  [ 6,  7,  8, 15, 16, 17]]
```

8.4 NumPy vs MATLAB

As linguagens Python e MATLAB/Octave tem diferenças que devem ser observadas ao programar. Embora a tradução de código seja quase que transparente, alguns detalhes podem fazer a diferença de performance e uso de memória nas plataformas, além de poderem introduzir comportamento inesperado.

Tabela 5 – Diferenças Básicas

MATLAB	Python
O principal elemento é um vetor multidimensional de elementos *double*. Operadores são feitos para serem operações matriciais.	O tipo principal da NumPy é o objeto **array**. Operações nestes objetos são por padrão elemento por elemento.
Indexação começa em 1.	Indexação começa em 0.
Linguagem tem passagem por valor, com um esquema de lazy-copy-on-write que previne a criação de cópias antes que elas sejam realmente necessárias. Operações *slice* são cópias do *array* original.	Linguagem tem passagem por referência. Operações *slice* são *views* do array original.

8.4.1 Equivalência de funções

A Tabela 6 mostra uma relação dos comandos mais usados do MATLAB em Python/NumPy.

Tabela 6 – Funções Equivalentes

help func	help(func)	Documentação da função
type func	source(func)	Código fonte da função (se não for nativa)
a && b	a and b	Operador lógico AND (apenas para escalares)
a || b	a or b	Operador lógico OR (apenas para escalares)
1*i, 1*j, 1i, 1j	1j	Números complexos

eps	np.spacing(1)	Distância entre 1 e o próximo número de ponto flutuante.
ode45	scipy.integrate.ode(f) .set_integrator('dopri5')	Integra uma EDO com Runge-Kutta 4,5
ode15s	scipy.integrate.ode(f) .set_integrator('vode', method='bdf', order=5)	Integra uma EDO com BDF
ndims(a)	ndim(a) ou a.ndim	Número de dimensões de um array
numel(a)	size(a) ou a.size	Número de elementos de um array
size(a)	shape(a) ou a.shape	Tamanho da matriz
size(a,n)	a.shape[n-1]	Número de elementos na n-ésima dimensão
[1 2 3; 4 5 6]	array([[1.,2.,3.], [4.,5.,6.]])	Matriz 2x3
[a b; c d]	vstack([hstack([a,b]), hstack([c,d])]) ou bmat('a b; c d').A	Constrói uma matriz das matrizes a, b, c e d
a(end)	a[-1]	Último elemento da matriz a
a(2,5)	a[1,4]	Elemento na segunda linha, quinta coluna
a(2,:)	a[1] ou a[1,:]	Segunda linha de a
a(1:5,:)	a[0:5] ou a[:5] ou a[0:5,:]	Primeiras 5 linhas de a
a(end-4:end,:)	a[-5:]	Últimas 5 linhas de a
a(1:3,5:9)	a[0:3][:,4:9]	Linhas 1 até 3 e colunas 5 até 9 de a. Acesso somente leitura.

a([2,4,5],[1,3])	a[ix_([1,3,4],[0,2])]	Linhas 2, 4 e 5, e colunas 1 and 3.
a(3:2:21,:)	a[2:21:2,:]	Linhas intermitentes de a, começando da terceira e indo até a vigésima primeira.
a(1:2:end,:)	a[::2,:]	Linhas intermitentes, começando da primeira.
a(end:-1:1,:) ou flipud(a)	a[::-1,:]	Linhas em ordem invertida
a([1:end 1],:)	a[r_[:len(a),0]]	a com uma cópia da primeira linha concatenada no final.
a.'	a.transpose() ou a.T	Transposta de a
a'	a.conj().transpose() ou a.conj().T	Transposta conjugada de a
a * b	a.dot(b)	Multiplicação matricial
a .* b	a * b	Multiplicação elemento-a-elemento
a./b	a/b	Divisão elemento-a-elemento
a.^3	a**3	Exponenciação elemento-a-elemento
(a>0.5)	(a>0.5)	Matriz onde o elemento (i,j) é ($a_{i,j} > 0.5$). Em MATLAB 0 e 1, em Python **True** e **False**
find(a>0.5)	nonzero(a>0.5)	Índices onde $a > 0.5$

a(:,find(v>0.5))	a[:,nonzero(v>0.5)[0]]	Extrai as colunas de a onde $v > 0.5$
a(:,find(v>0.5))	a[:,v.T>0.5]	Extrai as colunas de a onde o vetor coluna $v > 0.5$
a(a<0.5)=0	a[a<0.5]=0	a com os elementos menores que 0.5 zerados
a .* (a>0.5)	a * (a>0.5)	a com os elementos menores que 0.5 zerados
a(:) = 3	a[:] = 3	Muda todos os valores para o mesmo escalar
y=x	y = x.copy()	NumPy atribui por referência
y=x(2,:)	y = x[1,:].copy()	NumPy slices são referências
y=x(:)	y = x.flatten()	Transforma matrix em vetor
1:10	arange(1.,11.) ou r_[1.:11.] ou r_[1:10:10j]	Cria um vetor
0:9	arange(10.) ou r_[:10.] ou r_[:9:10j]	Cria um vetor
[1:10]'	arange(1.,11.)[:, newaxis]	Cria um vetor coluna
zeros(3,4)	zeros((3,4))	Matrix 3x4 cheia de zeros (64-bit ponto flutuante)
zeros(3,4,5)	zeros((3,4,5))	Matrix 3x4x5 cheia de zeros (64-bit ponto flutuante)
ones(3,4)	ones((3,4))	Matrix 3x4 cheia de ums (64-bit ponto flutuante)

eye(3)	eye(3)	Matrix identidade 3x3
diag(a)	diag(a)	Vetor dos elementos diagonais de a
diag(a,0)	diag(a,0)	Matrix diagonal quadrada onde os elementos não-zero são os elementos de a
rand(3,4)	random.rand(3,4)	Matrix 4x4 aleatória
linspace(1,3,4)	linspace(1,3,4)	Vetor com 4 elementos igualmente espaçados entre 1 e 3 inclusivo
[x,y] = meshgrid(0:8, 0:5)	mgrid[0:9.,0:6.] ou meshgrid(r_[0:9.],r_[0:6.]	Dois vetores 2D: um com x e outro com y
	ogrid[0:9.,0:6.] ou ix_(r_[0:9.],r_[0:6.]	Melhor maneira de executar função em uma grid
[x,y] = meshgrid([1,2,4], [2,4,5])	meshgrid([1,2,4],[2,4,5])	
	ix_([1,2,4],[2,4,5])	Melhor maneira de executar função em uma grid
repmat(a, m, n)	tile(a, (m, n))	Cria m por n cópias de a
[a b]	concatenate((a,b),1) ou hstack((a,b)) ou column_stack((a,b)) ou c_[a,b]	Concatena colunas de a e b
[a; b]	concatenate((a,b)) ou vstack((a,b)) ou r_[a,b]	Concatena linhas de a e b
max(max(a))	a.max()	Maior elemento de a
max(a)	a.max(0)	Maior elemento de cada coluna de a

max(a,[],2)	a.max(1)	Maior elemento de cada linha de a
max(a,b)	maximum(a, b)	Compara a e b elemento por elemento e retorna o valor máximo de cada par
norm(v)	sqrt(dot(v,v)) ou np.linalg.norm(v)	Norma L2 do vetor v
a & b	logical_and(a,b)	Operador AND elemento por elemento
a | b	logical_or(a,b)	Operador OR elemento por elemento
bitand(a,b)	a & b	Bitwise AND
bitor(a,b)	a | b	Bitwise OR
inv(a)	linalg.inv(a)	Inversa da matriz quadrada a
pinv(a)	linalg.pinv(a)	Pseudo-inversa da matriz a
rank(a)	linalg.matrix_rank(a)	Rank da matriz a
$\underline{a}$	linalg.solve(a,b) if a is square; linalg.lstsq(a,b) otherwise	Solução de $a * x = b$ para x
b/a	Solve a.T x.T = b.T instead	Solução de $x * a = b$ para x
[U,S,V]=svd(a)	U, S, Vh = linalg.svd(a), V = Vh.T	Singular Value Decomposition de a
chol(a)	linalg.cholesky(a).T	Fatorização Cholesky de uma matrix (superior no MATLAB, inferior no Python)
[V,D]=eig(a)	D,V = linalg.eig(a)	Autovalores e Autovetores de a

[V,D]=eig(a,b)	V,D = np.linalg.eig(a,b)	Autovalores e Autovetores de a, b
[V,D]=eigs(a,k)		Maior k autovalor e autovetor de a
[Q,R,P]=qr(a,0)	Q,R = scipy.linalg.qr(a)	Decomposição QR
[L,U,P]=lu(a)	L,U = scipy.linalg.lu(a) ou LU,P=scipy.linalg.lu_factor(a)	Decomposição LU (nota: P(Matlab) == transposta(P(numpy)))
conjgrad	scipy.sparse.linalg.cg	Solver de gradiente conjugado
fft(a)	fft(a)	Transformada de Fourier de a
ifft(a)	ifft(a)	Transoformada inversa de Fourier de a
sort(a)	sort(a) ou a.sort()	Ordena a matriz a
[b,I] = sortrows(a,i)	I = argsort(a[:,i]), b=a[I,:]	Ordena as linhas da matriz a
regress(y,X)	linalg.lstsq(X,y)	Regressão multilinear
decimate(x, q)	scipy.signal.resample(x, len(x)/q)	Downsample com filtro passa-baixa
unique(a)	unique(a)	
squeeze(a)	a.squeeze()	
all(a)	all(a)	True se todos os elementos forem verdadeiros
any(a)	any(a)	True se qualquer elemento for verdadeiro
polyval(p)	polyval(p)	

As funções de integração do Python funcionam de forma diferente das funções MATLAB. O exemplo abaixo define uma função **ode45** que funciona de forma mais parecida com a função do MATLAB e pode servir de base para aplicações mais avançadas.

```
1  import scipy.integrate
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  def ode45(f, interval, initial_values, points=100):
6      solver = scipy.integrate.ode(f)
7      solver.set_integrator('dopri5')
8      solver.set_initial_value(initial_values) # [y0, t0]
9      integrate = np.frompyfunc(solver.integrate, 1, 1)
10     if len(initial_values) == 1:
11         return np.hstack(integrate(np.linspace(*interval, points)).flat)
12     else:
13         return np.hstack(integrate(np.linspace(*interval, points)).flat).reshape(points, 2)[:, 0]
14
15 # a * \dot{y} + b * y = c * sin(x)
16 A = 0.4
17 B = 0.6
18 h0 = 0.3
19 delta = -0.05
20
21 x = np.linspace(0, 10, 100)
22 y = ode45(lambda t, h: (B * h0 + delta - B * h) / A, [0, 10], [0])
23 plt.plot(x, y, '-r')
24
25 # y +2y +2y=cos(2x),y(0)=0,y (0)=0
26 # z y z +2z+2y=cos(2x),z(0)=y(0)=0
27 x = np.linspace(0, 10, 100)
28 y = ode45(lambda t, y: [y[1], -2 * y[1] - 2 * y[0] + np.cos(2 * t)], [0, 10], [0, 0])
29 plt.plot(x, y, '-b')
30
31 plt.show()
```

8.4.2 Sistemas Lineares

Para resolver sistemas lineares a função **numpy.linalg.solve** pode ser utilizada:

Sistema de equações:

$$\begin{cases} 3x + 5y + z = 1 \\ 7x – 2y + 4z = 2 \\ -6x + 3y + 2z = 0 \end{cases} \tag{8.1}$$

Forma matricial:

$$\begin{vmatrix} 3 & 5 & 1 \\ 7 & -2 & 4 \\ -6 & 3 & 2 \end{vmatrix} * \begin{vmatrix} x \\ y \\ z \end{vmatrix} = \begin{vmatrix} 1 \\ 2 \\ 0 \end{vmatrix} \tag{8.2}$$

```
1 import numpy as np
2
3 A = np.array([[3, 5, 1], [7, -2, 4], [-6, 3, 2]])
4 b = np.array([1, 2, 0])
5
6 print(np.linalg.solve(A, b)) # [0.13100437, 0.06113537, 0.30131004]
```

8.4.3 Expressões Simbólicas

Expressões simbólicas são compostas por números e variáveis matemáticas indefinidas. Utilizando o pacote **sympy** é possível definir incógnitas matemáticas para usar em expressões. Isto é útil para descrever e manipular algébricamente funções e expressões.

Para definir uma variável simbólica use a função **symbols**:

```
1 import sympy as sp
2
3 t = sp.symbols('t')
4 x = sp.symbols('x(t)')
5 y, s = sp.symbols('y(t) s')
6 dy = sp.diff(y(t), t)
7
8 edo = sp.Eq(dy + 3*y, x)
9 sp.solve(edo, dy)
```

8.4.3.1 Somatório

A função **summation** faz somatórios.

```
import sympy as sp

x, n = sp.symbols('x(t) n')

sp.summation(x**n/sp.factorial(n), (n, 0, sp.oo)) # exp(x(t))
```

8.4.3.2 Limite

A função **limit** calcula limites.

```
from sympy import limit, sin, Symbol, oo
from sympy.abc import x

limit(sin(x)/x, x, 0) # 1
limit(1/x, x, 0, dir="+") # oo
limit(1/x, x, 0, dir="-") # -oo
limit(1/x, x, oo) # 0
```

8.4.3.3 Derivada

A função **diff** calcula a n-ésima derivada.

```
from sympy import sin, cos, Function, diff
from sympy.abc import x, y
f = Function('f')

diff(sin(x), x) # cos(x)
diff(f(x), x, x, x) # Derivative(f(x), x, x, x)
diff(f(x), x, 3) # Derivative(f(x), x, x, x)
diff(sin(x)*cos(y), x, 2, y, 2) # sin(x)*cos(y)
```

8.4.3.4 Integral

A função **integrate** integra uma função.

```
from sympy import integrate, log, exp, oo
from sympy.abc import a, x, y

integrate(x*y, x) # x**2*y/2
integrate(log(x), x) # x*log(x) - x
integrate(log(x), (x, 1, a)) # a*log(a) - a + 1
integrate(x) # x**2/2
```

8.4.3.5 Manipualação de Expressões

Os comandos **simplify**, **expand**, **factor**, **collect**, **cancel** e **apart** podem ser utilizados para manipular expressões algébricas.

```
import sympy as sp
from sympy.abc import x, y

sp.simplify( sp.cos(x)**2 + sp.sin(x)**2 ) # 1
sp.expand( (x + 2)*(x - 3) ) # x**2 - x - 6
sp.factor( x**3 - x**2 + x - 1 ) # (x - 1)*(x**2 + 1)
sp.collect( x**2 + 2*(x + y) - x*y, x) # x**2 + x*(-y + 2) + 2*y
sp.cancel( (x**2 + 2*x + 1)/(x**2 + x) ) # (x + 1)/x
sp.apart(  (3*x+5)/(1-2*x)**2 ) # 3/(2*(2*x - 1)) + 13/(2*(2*x - 1)**2)

exp = (x + 2)*(x - 3)
exp.subs(x, 3) # 0
sp.roots(exp) # {3: 1, -2: 1}
sp.solve(exp, x) # [-2, 3] | resolve exp == 0
```

8.4.3.6 Arquivos MAT

É possível salvar e carregar aquivos .MAT no Python de forma compatível com o MATLAB. Para isso usa-se o pacote **scipy.io** que contém as funções **loadmat**, **savemat** e **whosmat**.

```
import scipy.io as sio
import numpy as np

a = np.zeros( (2, 2) )
b = np.ones( (3, 3) )
c = np.eye(4)

sio.savemat('dados', {'a': a, 'b': b, 'c': c})
sio.whosmat('dados')
d = sio.loadmat('dados')
d['a'] == a
```

8.5 Matplotlib

A biblioteca **matplotlib.pyplot** permite a criação de gráficos usando uma interface parecida com a do MATLAB. As funções **plot** e **show** podem ser usadas para exibir gráficos. As funções **title** e **legend** também estão disponíveis.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x = np.linspace(-np.pi, np.pi, 256, endpoint=True)
5  c,s = np.cos(x), np.sin(x)
6
7  plt.figure(figsize=(10,6), dpi=120)
8
9  plt.plot(x, c, color='blue', linewidth=2.5, linestyle='-')
10 plt.plot(x, s, color='red',  linewidth=2.5, linestyle='-')
11 plt.legend(['Coseno', 'Seno'])
12 plt.title('Seno e Coseno')
13
14 plt.show()
15 plt.savefig('exemplo.eps', format='eps')
```

A função **plot** recebe como argumentos os pontos x, y e a formatação da linha. Também é possível passar argumentos nomeados para controlar a renderização do gráfico. A Tabela 8 mostra as formas de linhas disponíveis. A Tabela 7 mostra as cores reconhecidas e a Tabela 9 mostra alguns dos argumentos aceitos pela função **plot**.

Tabela 7 – Cores de linhas

'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

Tabela 8 – Tipos de linhas

-	solid line style	3	tri_left marker
–	dashed line style	4	tri_right marker
-.	dash-dot line style	s	square marker
:	dotted line style	p	pentagon marker
.	point marker	*	star marker
,	pixel marker	h	hexagon1 marker
o	circle marker	H	hexagon2 marker
v	triangle_down marker	+	plus marker
^	triangle_up marker	x	x marker
<	triangle_left marker	D	diamond marker
>	triangle_right marker	d	thin_diamond marker
1	tri_down marker	|	vline marker
2	tri_up marker	_	hline marker

Tabela 9 – Argumentos variáveis da função plot

alpha	0.0 -> transparente, 1.0 -> opaco
antialiased or aa	[True | False]
color or c	Qualquer cor da Tabela 7
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
label	Qualquer coisa que responda à formatação com **%s**
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '–' | '-.' | ':' | None | ' ' | '']
linewidth or lw	Número de ponto flutuante
rasterized	[True | False | None]
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
visible	[True | False]
zorder	Qualquer número

Capítulo 9

Otimização

Python é uma linguagem interpretada e sem compilador JIT (*Just In Time*). Por isso, usar algumas construções da linguagem podem fazer o programa ficar mais lento. Seguem algumas dicas de como melhorar o desempenho de seus programas.

Primeira regra da otimização: não otimize.

1. Faça certo.
2. Teste se está certo.
3. Profile se estiver lento.
4. Otimize.
5. Repita do 2.

9.1 Manipulação de strings

Strings são imutáveis.

```
# não faça:
s = ''
for substring in list:
    s += substring

# faça:
s = ''.join(list)
################################################################################
```

```
# não faça:
s = ''
for x in list:
    s += some_function(x)

# faça:
slist = [some_function(elt) for elt in somelist]
s = ''.join(slist)
```

Strings são imutáveis.

```
# não faça:
out = '<html>' + head + prologue + query + tail + '</html>'

# faça:
out = '<html>%s%s%s%s</html>' % (head, prologue, query, tail)

# ou:
out = '<html>%(head)s%(prologue)s%(query)s%(tail)s</html>' % locals()
```

9.2 Loops

Loops em Python são lentos, mas eles podem ser enviados para o interpretador, em C.

For < *List Comprehension* < **map**.

```
# não faça:
newlist = []
for word in oldlist:
    newlist.append(word.upper())

# faça:
newlist = map(str.upper, oldlist)

# ou (mas prefira o map):
newlist = [s.upper() for s in oldlist]
```

Todas as variáveis dentro do corpo do **for** e **while** são locais e devem ser carregadas em cada passagem do *loop*. Por isso é melhor criar variáveis fora do *loop* que aponte para as funções que serão usadas. Toda função chamada com um . será procurada em cada iteração.

```
upper = str.upper
newlist = []
append = newlist.append
for word in oldlist:
    append(upper(word))
```

O exemplo anterior pode ser posto fora de escopo (escopo global). Mas o acesso ao escopo local é muito mais rápido que ao escopo global. Colocar o bloco de código dentro de uma função ajuda a resolver este problema. A palava chave **include** coloca os pacotes no escopo global. Se uma função de um pacote for usada em um *loop*, referenciá-la no escopo local também pode tornar o acesso mais rápido.

```
def to_upper(oldlist):
    upper = str.upper
    newlist = []
    append = newlist.append
    for word in oldlist:
        append(upper(word))
    return newlist
```

9.3 Chamada de Função

Chamar funções em Python é lento, especialmente funções escritas em Python. Evite char uma função em um *loop* se esse bloco de código puder ser substituído por um *loop inline*. Essa dica vale especialmente para casos onde as bibliotecas SciPy/NumPy tem funções que operam em todo o vetor de uma vez, como **numpy.sin** e **numpy.exp**

```
# não faça:
def do1(x):
    return x + 1

for x in range(10):
    y = do1(x)

# faça:
for x in range(10):
    y = x + 1
```

9.4 Transferência de Conhecimentos

Python não é C. Nem Perl. Nem C++. Nem Java. Não assuma que coisas que funcionam em uma linguagem funcionará em Python. Ou em qualquer outra linguagem. Por exemplo, em C/C++ você pode declarar quantas variáveis quiser e usar numa chamada de função, que o código compilado será extamente o mesmo que passar os parâmetros diretamente pra função. Isso não é verdade em Python, onde todas as variáveis serão criadas e destruídas pelo interpretador.

```
a = 1
b = 2
c = a + b # Em C, isso é o mesmo que c = 1 + 2, e as variáveis a e b nunca serão criadas.
          # Na verdade, isso é o mesmo que c = 3 em C, já que o compilador fará essa conta.
          # Já em Python todas as variáveis serão criadas.
```

9.5 Try vs If

Seguindo o conceito de não carregar conhecimentos, temos algo curioso: na maioria das linguagens o bloco **try** tem um custo muito elevado. Não em Python. Na verdade, se você tem certeza que 99% das vezes a resposta de um **if** é a mesma, usar um **try** pode ser muito mais eficiente.

```
# não faça:
if somethingcrazy_happened:
    uhOhBetterDoSomething()
else:
    doWhatWeNormallyDo()

# faça:
try:
    doWhatWeNormallyDo()
except SomethingCrazy:
    uhOhBetterDoSomething()
```

9.6 NumPy

O uso das bibliotecas numéricas é normalmente mais rápido, já que elas são implementadas em C. No entanto há formas de melhorar a performance (ou evitar que ela piore

acidentalmente).

```
import numpy as np

a = np.arange(1, 13).reshape(4, 3)

# Use *= sempre que possível
a *= 2     # Nenhuma cópia criada
b = 2 * a  # Uma cópia de a é criada

# Alterar um array apenas causa uma cópia se a ordem do items for alterada
b = a.reshape( (4, 3) ) # Não causa cópia
b = a.T.flatten()       # Causa uma cópia

b = a.flatten() # Sempre causa uma cópia
c = a.ravel()   # Só se necessário
```

Há duas formas de acessar os valores de um **array**: *slicing* e com índices. Ambas as formas abaixo selecionam os elementos de 10 em 10. No entanto o formato com índices é bem mais lento. Uma alternativa é a função **take** se você precisa usar a forma de índices.

```
import numpy as np

n, d = 100000, 100
a = np.random.random_sample((n, d))

# slicing
b1 = a[::10]

# índice
b2 = a[np.arange(0, n, 10)]

# take
b3 = np.take(a, np.arange(0, n, 10), axis=0)
```

9.7 Vetorização

Vetorizar é transformar um código (normalmente um *loop*) em uma expressão que envolva operações matriciais. Isso faz uso do de instruções SIMD (*Single Instruction Multiple Data*) do processador e melhora a performance do programa.

```
1  import numpy as np
2
3  a = np.arange(10).reshape(10, 1)
4
5  # somatória dos quadrados com loop:
6  s = 0
7  for i in a.ravel():
8      s += i ** 2
9
10 # ou
11 s = sum(a.ravel() ** 2)
12
13 # versão vetorizada
14 s = a.T @ a
```

Bibliografia

[1] C. Rossant, *IPython Cookbook*, jul. de 2017. endereço: `http://ipython-books.github.io/featured-01`.

[2] M. Agarwal, *Essential Python Code Optimization Tips and Tricks for Geeks*, jul. de 2017. endereço: `http://www.techbeamers.com/python-code-optimization-tips-tricks`.

[3] "James" e "brett", *Optimizing Python Code*, jul. de 2017. endereço: `https://stackoverflow.com/questions/7165465/optimizing-python-code`.

[4] *Performance Tips*, jul. de 2017. endereço: `https://wiki.python.org/moin/PythonSpeed/PerformanceTips`.

[5] *Quickstart tutorial*, jul. de 2017. endereço: `https://docs.scipy.org/doc/numpy-dev/user/quickstart.html`.

[6] *NumPy for Matlab users*, jul. de 2017. endereço: `https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html`.

[7] M. Foord, *Duck Typing in Python*, jul. de 2017. endereço: `http://www.voidspace.org.uk/python/articles/duck_typing.shtml`.

[8] *Python Object Oriented*, jul. de 2017. endereço: `https://www.tutorialspoint.com/python/python_classes_objects.htm`.

[9] Swaroop, *Object Oriented Programming*, jul. de 2017. endereço: `https://python.swaroopch.com/oop.html`.

[10] *Data Structures*, jul. de 2017. endereço: `https://docs.python.org/3/tutorial/datastructures.html`.

[11] *Python Lists*, jul. de 2017. endereço: https://www.tutorialspoint.com/python/python_lists.htm.

[12] *Python Functions*, jul. de 2017. endereço: https://www.tutorialspoint.com/python/python_functions.htm.

[13] *Learn Python Programming*, jul. de 2017. endereço: https://www.programiz.com/python-programming.