Python

Este resumo é velhinho, mas pode ser útil.
Por Viviane Maranhão

Interpretador:

Para sair: CTRL+D ou "import sys; sys.exit()".
Modo interativo:
-prompt primário: >>>
-prompt secundário: ... (para terminar comandos de múltiplas linhas usar uma linha em branco)
Comentários: #
É possível fazer múltiplas atribuições: `a, b, c = 1, 2 ,3`
Impressão: `print`. Funciona como o printf do C, mas sem precisar especificar o tipo da variável. Vírgula no final faz não imprimir quebra de linha (num loop, por exemplo)

Números:

Atribuição: = (É possivel atribuir valor para mais de uma variável por vez. *Ex: x = y = z = 0*)
Divisão de inteiros retorna a parte inteira inferior
Para usar float basta escrever o número com o ponto flutuante.
Operações de tipos diferentes convertem os inteiros em float
Números complexos: a parte imaginária é representada por j ou J. Eles também podem ser criados através de: complex(real, imag). Se z é um complexo, é possivel obter sua parte real e imaginária através de z.real e z.imag. Os complexos são sempre representados com dois floats. abs(z) retorna um float com a norma de z
Para utilizar a última expressão impressa: _

Strings:

Podem vir entre aspas simples ou duplas.
Para imprimir uma string str1 usar: `print str1`
Strings entre aspas (simples ou duplas) triplas (”””) não precisam conter caracteres de quebra de linha
Concatenação: +
Repetir a string n vezes: `palavra*n`
Acessar posição n da palavra: `palavra[n]`. Usar um número negativo conta a partir da direita
Acessar intervalo da palavra: `palavra[x:y]`
Acessar n primeiros caracteres: `palavra[:n]`
Acessar tudo menos os n primeiros caracteres: `palavra[n:]`
Acessar n últimos caracteres: `palavra[-n:]`
Número de caracteres da palavra: `len(palavra)`
Não é possivel modificar uma posição da string
Converter para string: `str()`
Esquema de contagem de posições

```
 +---+---+---+---+---+
 | T | e | s | t | e |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Criar strings em Unicode: colocar “u” na frente do texto

Listas

Usada para agrupar dados de tipos diferentes. Separar por vírgula:

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

É possível mudar um elemento de uma lista.
O acesso às posições é como nas strings
Adicionar elementos na posição n: `lista[n:n] = [<conteúdo>]`
Remover elementos: `lista[x:y] = []`
Substituir elementos: `lista[x:y] = [conteúdo]`
Inserir uma copia da lista na posição n: `lista[:n] = lista`
Esvaziar a lista: `lista[:] = []`
Número de elementos da lista: `len(lista)`
É possível criar uma lista contendo uma lista
Métodos para objetos lista:
`append(x)`: Adiciona um item no final da lista (`lista[len(lista):] = [x]`)
`extend(L)`: Adiciona uma lista no final da lista (`lista[len(lista):] = L`)
`insert(i, x)`: Insere um item x na posição i
`remove(x)`: Remove o primeiro item da lista que tiver valor x (ele deve existir)
`pop([i])`: Remove o item de posição i da lista e o retorna. Se i não for especificado remove o último
`index(x)`: Retorna o índice da primeira ocorrencia de x (Deve existir x na lista)
`count(x)`: Retorna o número de vezes que x aparece na lista
`sort()`: Ordena os elementos da lista
`reverse()`: Ordena inversamente os elementos da lista
Listas como pilhas: Usar `append(x)` e `pop()`
Listas como filas: Usar `append(x)` e `pop(0)`

Controle de fluxo:

Terminam com uma linha em branco

Operadores de condição: os mesmos de C, is, isnot, in, notin. Comparações podem ser encadeadas. Operadores booleanos: or, and, negados com not. Os argumentos são avaliados da esquerda para a direita e pára quando a saída é determinada.

Zero é false, outros inteiros são true. O mesmo vale para strings (vazia é falsa)

É possível comparar objetos de seqüências (convém que com um objeto com o mesmo tipo de seqüência). Comparação lexicográfica

- while:

```
while <cond>:
      ... (é importante dar o tab antes do comando.)
```

- if:

```
if <cond>:
      ...
elif <cond>:
      ...
else:
      ...
```

- for: O for percorre listas ou strings.

```
>>> # Medindo strings
... a = ['cat', 'window']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
```

- break: Sai do loop do for o while mais próximo

- continue: Continua com a próxima iteração do loop

- else: Executado quando o loop termina por exaustão (for) ou quando a condição se torna falsa (while) e não houver um break.

- pass: Não faz nada

- range: cria sequências de números: `range(inicial, final+1, passo).` Pode assumir valores negativos. Pode ser usado para percorrer listas: `for i in range(len(lista)): ...`

Técnicas de looping

`iteritems()`: método para obter a chave e o valor de um item de dicionário em um loop

`enumerate(seq)`: função que imprime a posição dos elementos junto com os mesmo numa seqüência:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

`zip(seqs)`: função para trabalhar com duas ou mais seqüências em um loop:

```
>>> questions = ['Abacate', 'Melão']
>>> answers = ['verde', 'amarelo']
>>> for q, a in zip(questions, answers):
...     print 'Qual é a cor do %s?  É %s.' % (q, a)
...
Qual é a cor do Abacate?  É verde.
Qual é a cor do Melão?  É amarelo.
```

`reversed(seqüência)`: Trabalha com a seqüência na ordem inversa

`sorted(seqüência)`: Trabalha com a seqüência ordenada

Tuplas:

Uma tupla é uma seqüência de itens separados por vírgulas:

```
>>>t = 12345, 54321, 'bla'
>>>t
(12345, 54321, 'bla')
```

Elas sempre vêm entre parenteses, permitindo criar uma tupla de tuplas. São imutáveis como strings, mas é possível criar tuplas de objetos mutáveis como listas.

Uma tupla vazia é criada com um par de parenteses vazio e a de um elemento é criada com o elemento seguido de uma vírgula.

É possível desagrupar esses valores. (também funciona para outras sequências): `x, y, z = t`

Funções:

`def <nome>(<parametros>):`

A primeira linha da função pode ser uma string de documentação (docstring)

É possivel renomear funções: `nome_novo = nome_func`

Elas podem retornar valores (usar return)

É possivel usar valores default nos parâmetros (palavra chave): `def f(x, y z = <valor>)` que são atribuídos no momento da definição da função. Esse valor só é avaliado uma vez, faz diferença se ele é um objeto mutável como uma lista:

```
>>>def f(a, L=[]):
...   L.append(a)
...   return L
...
>>>print f(1)
[1]
>>>print f(2)
[1, 2]
>>>print f(3)
[1, 2, 3]
```

É possível fazer de outra maneira, para a lista não ficar crescendo:

```
>>>def f(a, L=None):
...    if L is None:
...        L = []
...    L.append(a)
...    return L
>>>print f(1)
[1]
>>>print f(2)
[2]
```

A função também pode ser chamada especificando as palavras chave: `f(6, 8, z = <valor>)`

`*<nome>`: indica um número não especificado de argumentos

`**<nome>`: indica um número nao especificado de palavras chave (dicionário)

Da mesma maneira é possível passar como argumento valores agrupados em uma lista:

```
>>> args = [3, 6]
>>> range(*args)
```

Forma lambda: Pequenas funções de uma expressão que servem quando é necessário um objeto função:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Funções úteis:

`filter(função, seqüência)`: retorna uma seqüência com os itens para os quais `função(item)` é verdadeira:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

`map(função, seqüência)`: chama função item para cada termo da seqüência e retorna uma lista com os resultados. É possível passar mais de uma sequência por vez:

```
>>> seq = range(8)
>>> def add(x, y): return x + y
...
>>>map (add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce (função, seqüência)`: retorna um valor obtido através da função binária "função" no primeiros dois itens da seqüência, então o resultado no próximo, até o fim da seqüência. É possível passar um terceiro argumento como valor inicial.

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

Abrangência de Lista (List Comprehensions): Expressão seguida de um for, e então, não necessariamente, mais fors e ifs. O resultado é uma lista. Se a expressão gerar uma tupla, a mesma deve ser inserida entre parênteses:

```
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [3*x for x in vec if x > 3]
[12, 18]
```

`del`: Remove uma variável, um, alguns ou todos os itens de uma variável

Conjuntos

Coleção sem elementos repetidos. Suportam operações matemáticas:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)               # cria um conjunto sem os duplicados
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # testando os elementos
True
>>> 'crabgrass' in fruit
False
>>> # Operacoes com conjuntos
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                             # letras em a mas nao em b
set(['r', 'd', 'b'])
>>> a | b                             # letras em a ou b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                             # letras em a e b
set(['a', 'c'])
>>> a ^ b                             # letras em a ou b mas nao nos dois
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Dicionários:

Indexados por chaves únicas que podem ser quaisquer elementos imutáveis (numero, string, tupla sem elementos mutáveis). Segue a seguinte estrutura: `{ chave1:valor, chave2:valor, ... }`
Um par de chaves cria um dicionário vazio

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127      # adiciona uma chave
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']          # apaga uma chave
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()               # retorna as chaves do dicionario
['guido', 'irv', 'jack']
>>> tel.has_key('guido')     # verifica se contem a chave
True
>>> 'guido' in tel           # verifica se contem a chave
True
```

`dict()`: cria um dicionário. Diferentes tipos de sintaxe:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # usa uma abrangencia de lista
{2: 4, 4: 16, 6: 36}
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Módulos:

Contém a extensão .py. Para utilizar basta dar o comando: `import <arquivo>` (sem .py).
Para utilizar as funções contidas nesse arquivo, após importado, é preciso usar o nome do modulo: `nome.func`.
A variável global `__name__` contém o nome do módulo.
A variável global `__buidin__` contém os nomes de funções ou variáveis internas.
Para importar funções diretamente: `from <arq> import <func1>, <func2> ...`
Para importar todos os nomes (menos os que começam com _) definidos em um módulo: `from <arq> import *` (prática não recomendada)
`dir(<modulo>)`: mostra quais nomes o módulo define. Em branco mostra os nomes atuais
Existem alguns módulos padrão. Um importante é o sys. Suas variáveis ps1 e ps2 definem as strings que serão usadas nos prompts primário e secundário
Ao importar um arquivo, o interpretador primeiro busca no diretório corrente, depois na variável de ambiente PYTHONPATH e por fim na lista de caminhos especificados em sys.path.
Para modificar este arquivo;

```
>>> import sys
>>> sys.path.append('/caminho')
```

Arquivos compilados:

São arquivos .pyc originados de um .py
Invocar o compilador com -O gera arquivos compilados otimizados .pyo
Programas compilados são indepentendes do .py e levam menos tempo para carregar, mas levam o mesmo tempo de execução que um .py
`compileall`: compila todos os módulos de um mesmo diretório.

Pacotes:

É uma forma de organizar módulos utilizando pontos.

É preciso que o primeiro diretório (que pode contém outros diretórios com módulos) contenha um arquivo `__init__.py` que pode ser vazio, inicializar algum pacote ou gerar a variável `__all__` que contém a lista dos nomes dos módulos a serem importados quando o comando: `from <pacote> import *` é encontrado.

Pacotes suportam mais um atributo especial, `__path__`. Este é inicializado como uma lista contendo o nome do diretório com o arquivo '`__init__.py`' do pacote, antes do código naquele arquivo ser executado. Esta variável pode ser modificada; isso afeta a busca futura de módulos e subpacotes contidos no pacote.

Entrada e Saída

`str()`: retorna uma representação de valor compreensível para pessoas

`repr()`: retorna uma representação para o interpretador

Números, listas, diciónarios entre outros valores têm a mesma representação nos dois casos. Strings e floats são distintos.

`rjust(n)`: método que justifica pela direita acrescentando os espaços necessários à esquerda. `ljust()` e `center()` são métodos similares a este.

`zfill(n)`: método que preenche uma string com zeros à esquerda até completar o tamanho n

`%`: usado para impor formatos à string:

```
>>> import math
>>> print 'O valor de pi é aproximadamente %5.3f.' % math.pi
O valor de pi é aproximadamente 3.142.
```

Se há mais do que um formato, então o argumento à direita deve ser uma tupla com os valores de formatação.

É possível também passar as variáveis por nome ao invés de por posição:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

`vars()`: mostra as variáveis locais

`open(<nome_arq>, <modo>)`: Abre um arquivo retornando um ponteiro para ele

`read(n)`: método que lê n bytes do arquivo (ou todo se n não for especificado)

`readline()`: método que lê uma linha do arquivo

`readlines()`: método que retorna uma lista com as linhas do arquivo

Outra mandeira de ler: `>>> for <linha> in <arq>:  print <linha>,`

`write(string)`: método que acrescenta uma string ao arquivo. Lembrar de converter para string

`tell()`: método que retorna um inteiro que indica a posição corrente de leitura ou escrita no arquivo, medida em bytes desde o início do arquivo.

`seek(offset, from_what)`: método para mudar a posição. A nova posição é computada pela soma de offset com from_what que pode assumir o valor 0 para indicar o início do arquivo, 1 para indicar a posição corrente e 2 para indicar o fim do arquivo. Este parâmetro pode ser omitido, quando é assumido o valor default 0.

`close(): métod que fecha o arquivo e libera recursos.`

`int()`: função que recebe uma string e converte seus valores para inteiro

Módulo pickle

Contém métodos que convertem strings para diversos objetos do python (listas, dicionários....) e vice versa

Se você possui um objeto qualquer x, e um objeto arquivo f que foi aberto para escrita, a maneira mais simples de utilizar este módulo é: `pickle.dump(x, f)`

Para desfazer, se f for agora um objeto de arquivo pronto para leitura: `x = pickle.load(f)`

Erros e excessões

```
>>> while True:
...     try:
...         x = int(raw_input("Entre com um número: "))
...         break
...     except ValueError:
...         print "erro"
...
```

Primeiro o programa tenta executar o que está entre `try` e `except`. Se não for gerada exceção, a cláusula `except` é ignorada e termina a execução da construção `try`. Se uma execução ocorre durante a execução da cláusula `try`, os comandos remanescentes na cláusula são ignorados. Se o tipo da exceção ocorrida tiver sido previsto junto àlguma palavra-reservada `except`, então essa cláusula será executada. Ao fim da cláusula também termina a execução do `try` como um todo.

A última cláusula `except` pode omitir o nome da exceção, servindo como uma máscara genérica.

É possível incluir uma clausula `finally` que sempre é executada após o `try`, mesmo havendo um `break` ou `return` dentro do `try`.

`raise`: permite ao programador forçar determinado tipo de excessão. O primeiro argumento é o nome da excessão e o segundo, opcional, é o argumento da excessão

Classes

- Espaço de nomes:

Mapeamento entre nomes e objetos. Não existe nenhuma relação entre nomes em espaços distintos.

Um escopo é uma região textual de um programa Python onde um espaço de nomes é diretamente acessível (uma referência sem qualificador especial permite o acesso ao nome).

- Sintaxe:

```
class NomeDaClasse(ClasseBase1, ClasseBase2): #Herança múltipla
    <comando-1>
    .
    .
    <comando-N>
```

Quando se fornece uma definição de classe um novo espaço de nomes é criado. Todas atribuições de variáveis são vinculadas a este escopo local. Em particular, definições de função também são armazenadas neste escopo

- Objetos de classe:

Quando termina o processamento de uma definição de classe, um objeto de classe é riado. Este objeto encapsula o conteúdo do espaço de nomes criado pela definição da classe.

Funções e variáveis são atributos de uma classe e são acessadas no objeto com ponto.

Instanciando uma classe: `x = NomeDaClasse()` atribui o objeto resultante (vazio) a variável local x

Quando uma classe define um método `__init__()`, o processo de instanciação automaticamente invoca `__init__()` sobre a recém-criada instância de classe. É possível também utilizar argumentos no método `__init__()`.

- Atributos:
Existem dois tipos de nomes de atributos válidos: atributos de dados e métodos.
Atributos de dados: Correspondem aos "menbros do" C++. É possivel criar novos atributos (de dados) para uma classe instanciada, eles passam a existir na primeira vez em que é feita uma atribuição. Atributos de dados sobrescrevem atributos métodos homônimos.
Métodos de objeto: Um método é uma função que "pertence a" uma instância. Em métodos o objeto (a qual o método pertence) é passado como o primeiro argumento da função. Em geral, chamar um método com uma lista de n argumentos é equivalente a chamar a função na classe correspondente passando a instância como o primeiro argumento antes dos demais argumentos. Quando um atributo de instância é referenciado e não é um atributo de dado, a sua classe é procurada. Se o nome indica um atributo de classe válido que seja um objeto função, um objeto método é criado pela composição da instância alvo e do objeto função. Quando o método é chamado com uma lista de argumentos, ele é desempacotado, uma nova lista de argumentos é criada a partir da instância original e da lista original de argumentos do método. Finalmente, a função é chamada com a nova lista de argumentos.
Frequentemente, o primeiro argumento de qualquer método é chamado `self` (convenção, pois `self` não possui nenhum significado especial em Python).

- Herança:
```
class NomeDaClasseDerivada(NomeDaClasseBase):
    <statement-1>
    .
    .
    <statement-N>
```
Se um atributo requisitado não for encontrado na classe, ele é procurado na classe base. Essa regra é aplicada recursivamente se a classe base por sua vez for derivada de outra. Classes derivadas podem sobrescrever métodos das suas classes base (corresponde ao "virtual" do C++).
Para estenter um método em uma classe derivada, ao invés de substituir o método sobrescrito de mesmo nome na classe base: `NoneDaClasseBase.nomedometodo(self, <argumentos>)`. Isso só funciona se a classe base for definida ou importada diretamente no escopo global.
Herança Múltipla: É feita uma busca em profundidade nas classes base da esquerda para a direita. É procurado na primeira classe base toda (e suas classes base) antes de partir para a próxima.

- Variáveis privadas:
Qualquer variável de `nome __var` (com pelo menos dois "_"no começo e no maximo um "_" no fim do nome) é automaticamente substituída por `_nomedaclasse__var`, onde nomedaclasse é o nome da classe corrente. Essa construção independe da posição sintática do identificador, e pode ser usada para tornar privadas: instâncias, variáveis de classe e métodos. Fora de classes, ou quando o nome da classe só tem '_', não se aplica esta construção.

- Geradores:
Geradores são uma maneira fácil e poderosa de criar iteradores. Eles são escritos como uma função normal, mas usam o comando `yield()` quando desejam retornar dados. Cada vez que `next()` é chamado, o gerador continua a partir de onde parou (sempre mantendo na memória os valores e o último comando executado):
```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
>>> for char in reverse('golf'):
...     print char,
...
flog
```

- Expressões geradoras:

Alguns geradores simples podem ser escritos sucintamente como expressões usando uma sintaxe similar a de abrangência de listas, mas com parênteses ao invés de colchetes. Essas expressões são destinadas a situações em que o gerador é usado imediatamente por uma função. Expressões geradoras tendem a ser muito mais amigáveis no consumo de memória do que uma abrangência de lista equivalente:

```
>>> sum(i*i for i in range(10))                    # soma dos quadrados
285
```

Biblioteca padrão

Argumentos da Linha de Comando: Esses argumentos são armazenados como uma lista no atributo argv do módulo sys:

Ex: foi digitado 'python demo.py um dois tres' na linha de comando:

```
>>> import sys
>>> print sys.argv
['demo.py', 'um', 'dois', 'tres']
```

Compressão de Dados: Formatos comuns de arquivamento e compressão de dados estão disponíveis diretamente através de alguns módulos, entre eles: `zlib, gzip, bz2, zipfile` e `tarfile`.

Internacionalização: Disponível através de módulos, como `gettext, locale`, e o pacote `codecs`.

`os`: Módulo que fornece dúzias de fúnções para interagir com o sistema operacional.

`shutil`: Para tarefas de gerenciamento diário de arquivos e diretórios, o módulo fornece uma interface de alto nível bem que é mais simples de usar

`glob:` Fornece uma função para criar listas de arquivos a partir de buscas em diretórios usando caracteres coringa (`*`).

`getopt`: Módulo que processa os argumentos passados em `sys.argv`. Outros recursos de processamento estão disponíveis no módulo `optparse`.

`re`: Fornece ferramentas para lidar processamento de strings através de expressões regulares. Ideal para reconhecimento de padrões complexos e manipulações elaboradas.

`math`: Oferece acesso às funções da biblioteca C para matemática e ponto flutuante

`random`: Fornece ferramentas para gerar seleções aleatórias

`urllib2`: Para efetuar download de dados a partir de urls

`smtplib:` Para enviar mensagens de correio eletrônico:

`datetime`: Fornece classes para manipulação de datas e horas nas mais variadas formas.

`timet`: Módulo usado para avaliar desempenho

`doctest`: Oferece uma ferramenta para realizar um trabalho de varredura e validação de testes escritos nas strings de documentação (docstrings) de um programa.

`unittest`: Não é tão simples de usar quanto o módulo `doctest`, mas permite que um conjunto muito maior de testes seja mantido em um arquivo separado

`xmlrpclib` e **`SimpleXMLRPCServer`**: Para a implementação de chamadas remotas

`email`: É uma biblioteca para gerenciamento de mensagens de correio eletrônico, incluindo MIME e outros baseados no RFC 2822. Diferentemente dos módulos `smtplib` e `poplib` que apenas enviam e recebem mensagens, o pacote `email` tem um conjunto completo de ferramentas para construir ou decodificar estruturas complexas de mensagens (incluindo anexos) e para implementação de protocolos de codificação e cabeçalhos.

`xml.dom` e **`xml.sax`**: Oferecem uma implementação robusta deste popular formato de troca de dados.

`csv`: Permite ler e escrever diretamente num formato comum de bancos de dados.

`repr`: Oferece uma versão da função `repr()` customizada para abreviar a exibição contêineres grandes ou aninhados muito profundamente

`pprint`: Oferece controle mais sofisticada na exibição de tanto objetos internos quanto aqueles definidos pelo usuário de uma maneira legível através do interpretador.

`textwrap`: Formata parágrafos de texto de forma que caibam em uma dada largura de tela:

`locale`: Acessa um banco de dados de formatos de dados específicos à determinada cultura.

`string`: Inclui uma classe Template que permite customizar aplicações sem altera-las usando $

`struct`: Possui as funções `pack()` e `unpack()` para trabahar com registros binários de tamanho variado

`threading`: Módulo utilizado para trabalhar com multi-threading

`logging`: Oferece ferramentas para gerar logs

`weakref`: Oferece ferramentas para rastrear objetos sem criar uma referência. Quando o objeto não é mais necessário, ele é automaticamente removido de uma tabela de referências fracas e uma chamada (callback) é disparada.

`array`: Oferece um objeto `array()`, semelhante a uma lista, mas que armazena apenas dados homogêneos e de maneira mais compacta.

`collections`: Oferece um objeto `deque()` que comporta-se como uma lista com operações de anexação (`append()`) e remoção (`pop()`) pelo lado esquerdo mais rápidos, mas com buscas mais lentas no centro. Estes objetos são adequados para implementação de filas e buscas em amplitudade em árvores de dados Além de implementações alternativas de listas, a biblioteca também oeferece outras ferramentas como o módulo `bisect` com funções para manipulações de listas ordenadas

`heapq`: Oferece funções para implementação de heaps baseadas em listas normais. O valor mais baixo é sempre mantido na posição zero.

`decimal`: trabalha com a aritimética de ponto flutuante usando informações decimais

Mais informações:

Pyth12on Library Reference: http://www.python.org/lib/lib.html
Installing Python Modules: http://www.python.org/inst/inst.html
Language Reference: http://www.python.org/ref/ref.html
http://www.python.org
http://docs.python.org
http://cheeseshop.python.org
http://aspn.activestate.com/ASPN/Python/Cookbook/