Workshop de Python para Dados Biológicos 2017

INSTITUTO DE QUÍMICA - UNIV. DE SÃO PAULO

Versão 1.0 beta

Informações gerais e tutoriais das práticas

Instrutores
Mestrando Renato A. C. dos Santos (Universidade de Campinas)
Mestre Bruno F. de Souza (Universidade de São Paulo)
Graduando Alisson H. da Costa (Universidade Federal de São Carlos)
Mestranda Ana Ciconelle (Universidade de São Paulo)

Organização e Apoio
Mestrando Renato Augusto Corrêa dos Santos
Profa. Dra. Flávia Vischi Winck (Universidade de São Paulo)
Doutoranda Franciele G. Esteves (Universidade Estadual Paulista)
Mestre e Doutorando Leonardo Martins (Universidade Federal de São Paulo - Escola Paulista de Medicina)
Bacharel Raniere G. C. da Silva (Software Sustainability Institute, Universidade de Manchester, UK)

Contribuições adicionais (sugestões e revisão do trabalho)
Dr. rer. nat. Diego M. Riaño Pachón (Universidade de São Paulo) (palestrante)
Graduando André Calderán (Universidade Federal de São Carlos)
Graduando Filipe Albuquerque Russo (Universidade de São Paulo)

Instituto de Química, USP, SP
10 e 11 de Novembro de 2017

Patrocinadores

ThermoFisher SCIENTIFIC

illumina®

EXXTEND SOLUÇÃO EM OLIGOS

JB PESQUERO LABORATÓRIO

BECKMAN COULTER Life Sciences

Stoller

Universidade de São Paulo
Instituto de Química

Apoio


Universidade de São Paulo
Instituto de Química


Laboratório de
Biologia de
Sistemas
Regulatórios

Conteúdo

```
def parse_protein_table(fname):
    df = pd.read_table('proteins/{}'.format(fname))
    df['file_name'] = fname
    return df
df = parse_protein_table(files[0])
for f in files[1:]:
new_df = parse_protein_table(f)
    df = pd.concat([df, new_df])
df.to_csv('all_proteins.tsv', sep='\t', index=None)
```

Problema 2-2

```
import pandas as pd
from itertools import chain
import re
df = pd.read_excel('proteínas glândula aranha.xltx', skiprows=[0,1,2],
skip_footer=1)
texto_categorias = {}
texto_categorias[‘biological process’] = """biological adhesion, biological
regulation, cell killing, cellular process, developmental process, establishment of
localization, growth, immune system process, localization, locomotion, metabolic
process, multi-organism process, multicellular organismal process, response to
stimulus, rhythmic process, antioxidant activity, auxiliary transport protein activity,
binding, catalytic activity, chemorepellent activity, electron carrier activity,
enzyme regulator activity, molecular transducer activity, motor activity, nutrient
reservoir activity, structural molecule activity, transcription regulator activity,
translation regulator activity, transporter activity, chaperone regulator activity,
chemoattractant activity, metallochaperone activity, pigmentation, protein tag,
reproduction, reproductive process, transporter activity, viral reproduction"""
texto_categorias['cellular component'] = """Golgi apparatus, cytoplasm,
cytoskeleton, endoplasmic reticulum, endosome, extracellular region, intracellular
organelle, membrane, mitochondrion, nucleus, organelle membrane, organelle
part, plasma membrane, ribosome"""
texto_categorias['molecular function'] = 'molecular function'
categories = {}
For cat, texto in texto_categorias.items():
categories[cat] = re.split(r’,\s*’, texto)
colunas_subcategorias = list(chain(*categories.values()))
outras_colunas = [ ]
For col in df.columns:
If col not in colunas_subcategorias:
outras_colunas.append(col)
```

Apresentação

O workshop

O workshop é prático porque "**fluência computacional** é desenvolvida ativamente, não passivamente".

A fluência computacional exigida para o trabalho do biocientista hoje tem dois pilares:

- Conhecimento de uma linguagem de programação
- Compreensão de princípios centrais de ciência da computação

Diversos artigos recentes na área de Educação para Bioinformática e análise de dados por cientistas destacaram o papel de treinamento como facilitador da comunicação entre o cientista e especialistas em Bioinformática.

A ideia do workshop é introduzir uma linguagem de programação que tem muitas aplicações na área de Ciências Biológicas.

Motivação e surgimento da ideia

- Metodologias experimentais de alta performance, como a Genômica, a Transcriptômica, a Proteômica, a Metabolômica e outras "ômicas"
- Manipulação de longas tabelas, longos arquivos com sequências e anotações
- Limitação no treinamento e capacitação de cientistas na área de ciências biológicas, tanto para lidar com os dados como para interagir com bioinformata
- Iniciativas prévias, como os workshops da Software Carpentry e a publicação sobre Python para Biocientistas (Ekmekci et al. 2016)

Cronograma de Atividades

SEXTA-FEIRA (10 de Novembro de 2017)

7:30-8:15h Recepção dos participantes
- Solução de possíveis problemas de instalação de programas
- Entrega de senha para acesso à internet
- Entrega de kits

8:15-10h Introducão à linguagem Python
10-10:25h Coffee-break
10:25-12h Introdução à linguagem Python
12-13h Almoço
13-14h Palestra "Problemas biológicos e soluções computacionais"
(Dra. Flávia V. Winck e Dr. Diego M. Riaño Pachón)
14-15:25h Introdução à linguagem Python
15:25-15:50h Coffee-break
15:50-17:30h Introdução à linguagem Python
17:30-18:30h Brindes Exxtend
19:30h - Confraternização (Vila Butantan)

SÁBADO (11 de Novembro de 2017)

8-8:30h - Apresentação de problema 1: Dados tabulares de expressão gênica com RNAseq
8:30-10:15h - Apresentação de soluções para o problema 1 e exercícios
10:15-10:40h Coffee-break
10:40-12h Apresentação de soluções para o problema 1 e exercícios
12-13h Almoço
13-13:30h Apresentação de problema 2: Análise de dados de proteômica
13:30-15:15h Apresentação de soluções para o problema 2 e exercícios
15:15-15:40h Coffee-break (**formulário com feedback dos participantes**)
15:40-17:30h Apresentação de soluções para o problema 2 e exercícios
17:30h-18h Roda de bate papo sobre os problemas dos participantes

Primeiro Dia - Introdução à Linguagem Python

1. Fundamentos básicos de Programação

Sintaxe

Python é uma linguagem de programação e assim como na linguagem natural, os símbolos devem estar logicamente dispostos de acordo com o que é definido para formar uma palavra, frase, oração, etc. Na perspectiva de uma linguagem de programação, erros de sintaxe refere-se ao uso de instruções e expressões de modo que não foram definidas.

Exemplos:

- Abertura mas não fechamento de parênteses;
- Uso de símbolos especiais e espaços na declaração de variáveis e/ou funções;
- Não inserção do caracter ":" para delimitar o início de uma estrutura especial (como estruturas condicionais, de repetição, função, classes, etc.).

Semântica

Os erros semânticos, por sua vez, estão relacionados à lógica. Isto é, uma frase em linguagem natural pode estar gramaticalmente correta, porém, não fazer sentido.

O mesmo acontece com linguagens de programação, onde as instruções e expressões podem estar gramaticalmente corretas, entretanto, não realizam aquilo que o programador esperava.

Exemplos:

- Loops. Isto é, uma sequência de instruções que se repete infinitamente;
- Operações entre números e strings.

2. Python

Por que usar Python?

- Linguagem orientada a objetos e funcional
- É gratuita!
- É portátil!
- É poderosa!
- É simples de usar!
- É (relativamente) fácil de aprender!
- Linguagem de alto nível: tarefas de processamento de dados são realizadas de forma mais concisa.

Python em Biociências

- Semântica direta e sintaxe “limpa”
- Programação orientada a objetos
- Muitas bibliotecas disponíveis para aplicação em diversas áreas das Ciências Biológicas
    - Análises de sequências e estruturas
    - Filogenômica
    - E muito mais!

Exemplos de programas de Bioinformática

- ploidyNGS
- PyMOL
- Muitos outros

Python 2 ou 3 ?

Neste workshop usaremos Python 3.

Interpretador de Python

O interpretador Python é um programa que lê e executa um código Python linha a linha, ou seja, as **instruções**. Diferente dos compiladores, o interpretador verifica a sintaxe do código a cada linha e no caso de tudo estar correto, executa essa linha.

Linguagens interpretadas, como Python, são numericamente menos eficientes e mais lentas que linguagens de baixo nível como Assembly ou linguagens que usam compiladores como C, C++ e Java. Por outro lado, Python exige muito menos trabalho para programar.

Scripts

A maior parte dos códigos escritos em linguagens como Python e Perl no dia a dia são scripts. Por ser uma linguagem fácil de se programar, a criação de roteiros para sistemas computacionais não exige muito esforço ou complicações.

O prompt de comando e shells de Python

```
>>> print('Hello, world!')
>>> quit()
```

- IDLE

https://www.python.org/downloads/

- Jupyter

IDLE

O IDLE é uma interface gráfica para o interpretador de Python, permitindo a edição e execução de programas.

3. Jupyter notebook

O Jupyter notebook é uma ferramenta web open-source que permite criar e compartilhar documentos com códigos interativos, visualização de resultados, equações, além de texto livre.

O Jupyter integra o IPython, uma arquitetura para computação interativa em Python.

Rodando o Python no modo interativo

O prompt (IDLE e outras shells)

No modo interativo, o sinal “>>>” indica que o prompt está esperando uma entrada (comando) do usuário.

```
>>>
```

O resultado dos comandos deve aparecer abaixo deste sinal:

```
>>> print('Hello Word!')
Hello World
>>> print(2 ** 8)
256
>>> lumberjack = 'okay'
>>> lumberjack
'okay'
```

Rodando no modo interativo do Jupyter notebook

```
In []: print('Hello, world!')
```

No modo interativo não é necessário usar a função `print()` (funções serão detalhadas mais adiante) explicitamente para visualizar o conteúdo das variáveis.

```
>>> print('Hello Word!') # Usando uma função nativa de Python
```

Como em muitas outras linguagens, o sinal # indica um comentário. O Python não interpreta as caracteres que o seguem.

E por que usar o modo interativo?

- Testar programas
- Experimentar a linguagem (nosso caso!)

O diretório de trabalho

Verificando o diretório (pasta) de trabalho.

```
>>> import os
>>> os.getcwd()
```

Rodando códigos em arquivos (módulos/ programas/ scripts)

Desvantagem do prompt interativo: os comandos somem após terem sido interpretados.

Vamos usar o editor do **Jupyter** (poderíamos usar o gedit, bloco de notas, …)

```
# Primeiro código em Python
import sys
print(sys.platform)
print(2 ** 100)
x = 'Spam!'
print(x * 8)
```

Neste primeiro código, importamos um módulo de Python, roda a função print que mostra os resultados do script, usa a variável criada após tê-la criado e aplica métodos em objetos (discutidos a seguir).

.py é a terminação obrigatório para módulos (códigos importados em um programa de Python). É uma boa prática ***usar a extensão para todo programa escrito***.

Para rodar o código no Jupyter Notebook:

```
%run script.py
```

Usando módulos

Módulos são pacotes com ferramentas adicionais às providas por padrão pelo Python e que são armazenados em arquivos.

Para usar estes módulos, devemos importá-los em nosso código usando o comando `import`. Como exemplo, importamos abaixo o módulo `math`:

```
>>> import math
>>> math.pi
>>> math.sqrt(85)
```

Em Bioinformática, diversos módulos de Python já foram desenvolvidos para analisar sequências biológicas, estruturas de proteínas, ploidia de organismos com genomas sequenciados. Na página do Python Package Index há muitos outros.

Módulos padrão de Python

A biblioteca padrão de python apresenta diversos módulos úteis.

Alguns módulos:

- sys - parâmetros e operações de sistema
- os - interfaces com o sistema operacional

Exemplos de módulos úteis em Bioinformática incluem:

- re - operações envolvendo expressões regulares

Em geral, módulos que fornecem métodos para manipulação de strings são úteis, já que em Bioinformática trabalhamos muito com sequências (DNA, RNA, proteínas).

Importando módulos

A função `import` importa funções, classes, variáveis, etc:

Vamos criar o módulo **`meuarquivo.py`**:


```
titulo = 'amo muito tudo isso'
```

Vamos usar a variável importada do arquivo:

```
>>> import meuarquivo
>>> meuarquivo.titulo
```

Atributos geralmente estão seguidos de um ponto e estão associados a objetos em Python (uma das operações mais comuns no uso da linguagem).

```
>>> from meuarquivo import titulo
>>> titulo
```

4. Introdução à Programação com Python

Valores simples

```
>>> 9
>>> None
>>> Non # erro
```

Erros

Quando executamos algum código escrito em Python, o interpretador Python reporta quais os erros existentes no código que o impossibilitam de funcionar.

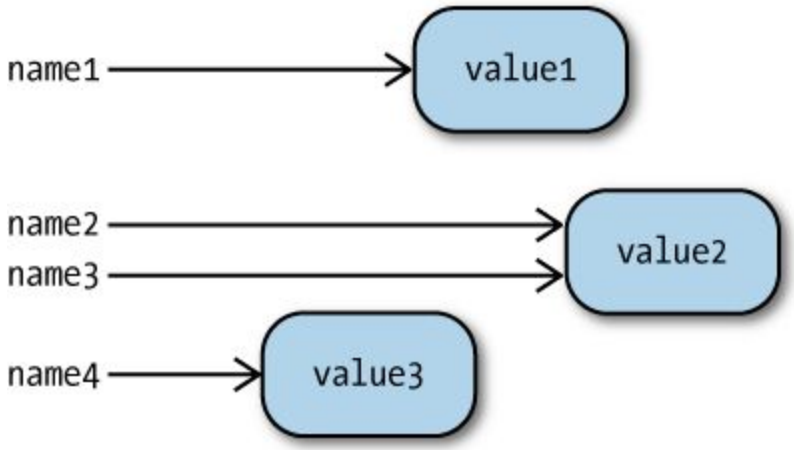
Exemplos:

- `NameError`: uso de variável antes de declará-la;
- `SyntaxError`: sintáxe inválida;
- `IndexError`: acessando ou atribuindo valor a um índice inexistente em uma sequência.
- `ModuleNotFoundError`: módulo não encontrado
- `FileNotFoundError`: O sistema não pode carregar o arquivo especificado
- `KeyError`:

Variáveis e Objetos

As variáveis são espaços de memória endereçados no seu programa capazes de armazenar algum tipo de dado, i.e, um valor específico. Um exemplo simples em Python é o armazenamento em uma declaração algébrica:

```
>>> x = 9
```



More than one name may refer to the same object

Fonte: Model, 2009

Em Python, as variáveis são objetos. Objetos fazem parte da programação orientada a objetos e contêm tanto dados como funcionalidade associados.

Objetos podem assumir várias formas como tipos básicos de dados ou estruturas complexas. Cada tipo de variável possui um domínio de valores que é capaz de armazenar e diversas operações associadas para sua manipulação.

Tipos básicos de dados

inteiros	20, 40, 123, 15859
reais (float)	45.12, 1.128, 3.14
complexos	1+2j, 5+3j, 10+7j
booleanos	True, False
strings (texto)	‘Python’, “Python”, ‘Py”thon”’, “Pyt’hon’”
nulo	None

Estruturas Complexas

Listas	[1, [2, ‘três’], 2.5], list((range(10))
Tuplas	(1, ‘spam’, 4, ‘U’)

Dicionários	{‘food’:’spam’,’taste’:’yum’}
Arquivos	open(‘eggs.txt’)
Sets	set(‘abc’),{‘a’,’b’,’c’}

Além disso, em Python uma variável só passa a existir após ser declarada, isto é, criada e alocado algum dado à ela e é importante que os nomes, não apenas de variáveis, mas também de funções, classes e outras estruturas que serão vistas posteriormente sejam sugestivos, para facilitar a compreensão do código.

```
>>> inteiro = 128 # variavel de numero inteiro
>>> real = 3.1415 # variavel de numero real
>>> complexo = 1+2j # variavel de numero complexo
>>> boolean = True # variavel booleana de valor verdeiro
>>> boolean = False # variavel booleana de valor falso
>>> string  = 'I Love Python'
>>> nulo    = None
```

Outra característica da linguagem Python (como de muitas outras linguagens de programação) é que a atribuição de valor para uma variável em determinado ponto do código sobrescreve qual atribuição feita anteriormente para a mesma variável.

```
>>> x = 2 # Primeira atribuição de valor à variável 'x'
>>> x = 5 # Segunda atribuição de valor à variável 'x'
>>> y = x + 2 # Uso da variável na atribuição de valor à variável 'y'
>>> x # Obtendo o valor de 'x'
```

Algumas **regras** para nomes de variáveis:

- Deve obrigatoriamente começar com letras
- Pode conter “underscore”/ sublinha e números
- Não deve ter espaço
- Python é “case sensitive”

Obtendo Ajuda

Obtendo a lista de atributos associados a um objeto

```
>>> dir(S)
```

Obtendo detalhes sobre métodos específicos:

```
>>> help(S.replace)
```

Funções

Uma vantagem de abordagens envolvendo programação é que o computador permite a **automatização de tarefas** (por exemplo, associadas a fluxos de trabalho para análise de dados). Funções são pequenos códigos que fornecem solução para um pequeno problema/tarefa. Outros partes do programa podem, então, "chamar" uma determinada função que retorna a solução para um problema desejado neste trecho.

Funções geralmente recebem argumentos como entrada e retornam um resultado baseado nestas entradas.

Argumentos podem ser fornecidos para função de forma dependente de sua posição dentro dos parênteses (argumentos posicionais) ou através do uso de palavras-chave ("keywords").

Nem todas as funções requerem argumentos (como a função `print()`)

```
>>> print()
```

Expressões com números

Objetos de Python que correspondem a números podem incluir inteiros, números com partes fracionárias (*floating-points*), números complexos, decimais com número fixo de casas após a vírgula, racionais.

Algumas operações matemáticas e operadores numéricos em Python:


```
>>> 123 + 222 # soma
>>> 123 - 222 # subtração
>>> 1.5 * 4 # multiplicação
>>> 20 / 10 # divisão
>>> 20 // 10 # divisão com resultado inteiro
>>> 4 % 5 # retorna apenas o resto da divisão
>>> 2 ** 100 # exponenciação
```

Além das operações nativas da linguagem, adicionais podem ser realizadas importante o módulo **`math`** e muitos outros disponíveis na internet.

Exercício 1 - Escreva um código que obtenha o número de ligações peptídicas para uma proteína com 20 aminoácidos.

Exercício 2 - Escreva um código que obtenha o número de resíduos de aminoácidos em uma proteína gerada, dado o tamanho da sequência codificante.

Operações lógicas

A compreensão de valores que seguem a "lógica booleana" é importante para compreender muitos aspectos em programação. Operadores lógicos (booleanos):

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

Atuando de forma complementar aos operadores booleanos no cotidiano do programador estão os operadores de comparação (ou relacionais).

```
>>> 1 < 4 < 6
>>> 2 < 2
>>> 2 > 5
```


```
>>> 'tc' in 'actcacactaac' # Veremos strings / cadeias de caracteres a seguir
>>> 'tc' in 'ACTCACACTAAC'
>>> 'tc'.upper() in 'ACTCACACTAAC'
```

Exercício 3 - Dadas duas variáveis que guardam os comprimentos de uma sequência de nucleotídeos codificante (CDS) e de uma proteína (valores numéricos inteiros), verificar se o tamanho da CDS condiz com o comprimento da proteína traduzida (fornecer comandos adicionais).

```
>>> compr_proteina = 30
>>> compr_cds = 90
>>> compr_proteina2 = 50
>>> compr_cds2 = 160
```

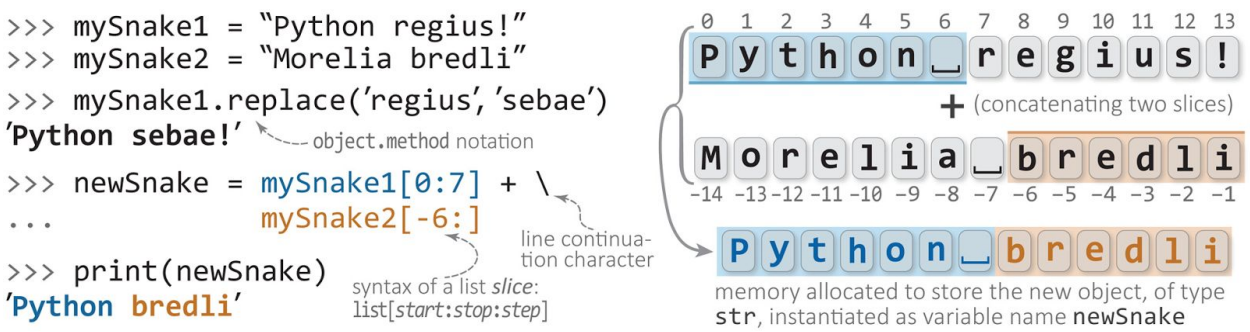
Voltaremos a discutir operações lógicas mais adiante (expressões condicionais).

Strings

Outro poderoso recurso em Python são as strings, elas armazenam informações textuais em geral e coleções arbitrárias de bytes (conteúdo de arquivos de imagens). Em Bioinformática strings são importantes, pois são usadas para representar sequências de DNA, RNA e proteínas.

É possível declarar uma string usando tantos aspas simples (‘ ’) quanto aspas duplas (“ ”). Isso permite com que você use aspas duplas no seu texto, caso tenha declarado a string com aspas simples, e vice-versa:

```
>>> meu_nome = 'Renato'
>>> python = 'I love Python'
>>> biologia = “Biologia e o maximo”
>>> cool = “Python’s cool”
```



Fonte: Ekmekci et al (2016)

Assim como em outras linguagens de programação, uma string em Python pode ser interpretada como uma cadeia de caracteres. Desse modo, é possível acessar tanto caracteres específicos em uma string como um intervalo de caracteres.
Para isso, utilizamos colchetes `[ ]` e dentro deles inserimos o índice do caractere que desejamos acessar. No caso, de um intervalo, inserimos os índices do primeiro e último caractere do intervalo separados por dois pontos `:` , ou como é chamado em Python, operador de slice.
Também é importante ressaltar que, os índices começam a partir de 0 e não de 1.

```
>>> python = 'I Love Python'
>>> python[0] = 'I' # primeira posicao é 0 e nao 1
>>> python[7:13] = 'Python' # usando o operador de slice
>>> python[-1] # Último item da string python
>>> python[-2] # Penultimo item da string python
```

Operações com Strings

Concatenação de strings usando o operador de soma/concatenação (+). O resultado das operações podem ser atribuídas a uma variável.

```
>>> python = 'python'
>>> python + ' baby' # var do tipo string + string
>>> python + python # var do tipo string + var do tipo string
>>> python + ' ' + python
>>> python + ' baby ' + python + ' baby'
>>> python[7:13] + ' baby'
>>> print(python + ' ' + 'baby')
```

Repetição de strings usando o operador de multiplicação/replicação (*). O resultado das operações podem ser atribuídas a uma variável.

```
>>> python * 10
>>> (python + ' ') * 10
>>> python[7:13] * 10
```

Exercício 4 - Obtenha um segmento de 3 aminoácidos a partir do vigésimo resíduo da sequência da seguinte proteína: 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'.

Métodos

Métodos são funções ligadas a objetos de tipos específicos. As operações realizadas acima com strings são métodos específicos para lidar com este tipo de objeto.

```
>>> S = 'spam'
>>> S.find('pa')
>>> S
>>> S.replace('pa','XYZ')
>>> S # A string continua inalterada, pois é um objeto imutável
```

As funções **`find`** e **`replace`** são exemplos de métodos aplicados a objetos do tipo string. Da mesma forma, há métodos aplicados a objetos de outros tipos, como números, listas, dicionários (detalhes a seguir).

Alguns métodos para strings

Há uma grande diversidade de funções/métodos para manipulação de strings. Os métodos mais comuns são:

index(): retorna o indice de primeira ocorrência do caractere um substring informada como parametro.

```
>>> python.index('Python')
```

lower() : retorna uma copia da string com todos os caracteres minusculos.

```
>>> python.lower()
```

upper() : retorna uma copia da string com todos os caracteres maisculos

```
>>> python.upper()
```

strip() : retorna uma copia da string sem o caractere ou substring informada como parâmetro. Se o parametro for vazio, são removidos os espaços da string.

```
>>> python.strip('I Love ')
```

Variações de strip() são lstrip() e rstrip().

replace() : retorna uma cópia da string com todas as ocorrencias de old substituidas por new.

```
>>> python.strip('Python', 'Biology')
```

split(): retorna uma lista com todos os caracteres da string, porém, sem str onde str é utilizado como separador da string. Também é possível delimitar o número de vezes em que a string deve ser dividida baseada na quantidade de vezes em que str for encontrado na string.

```
>>> python.split(' ')
```

Método para sequências: a função len()

Além de strings, outros objetos de Python são considerados sequências (ver glossário).

A função `len()` é uma das funções mais úteis e utilizadas na linguagem Python. Sua tarefa é retornar o tamanho (número de itens) de um objeto. Esses objetos podem ser strings, listas, tuplas, dicionários:

```
>>> len(python)
>>> python[7:len(python)] * 10
>>> python[len(python)-6:len(python)]
```

Método para sequências: a função str()

A função **`str()`** retorna uma representação em texto de um objeto, ou seja, retorna uma string que expressa o objeto enviado como parâmetro:

```
>>> int = 18446744073709551616
>>> len(str(int))
>>> len(str(18446744073709551616))
>>> len(str(2 ** 64))
```

Listas

Listas são coleções de objetos ordenados, sem comprimento fixo e mutáveis.

```
>>> L = []
>>> L = [123, 'spam', 1.23]
>>> len(L)
```

Muitas operações associadas a strings são também aplicáveis a listas, pois listas também são consideradas sequências.

```
>>> L[0]
>>> L[:-1]
>>> L + [3, 5, 6]
>>> L * 2
>>> L
```

Não há restrições de tipos de objetos incluídos em listas (veja que é heterogênea a lista acima: contém números, strings. Listas podem conter, inclusive, outras listas).

Alguns métodos aplicados a listas:

```
>>> L.append('NI')
>>> L
>>> L.pop(2)
>>> L
```

Tuplas

Tuplas são sequências, como strings e listas, mas são imutáveis.

```
>>> ('TCAG','UCAG')
>>>
>>>
```

Uma propriedade especial de listas e tuplas é que podemos iterar em seus elementos.

Iteração usando FOR

Realizando operações com cada elemento de uma lista com FOR:

```
>>> codons = ['UUU','UAU','UGA']
>>> for codon in codons:
  print(codon)
```

Dicionários

Dicionários são coleções do tipo “mapping”. Não são sequências (como listas e strings), mas também são coleções de outros objetos. Eles são organizados por chaves para acessar seus elementos e as posições não são ordenadas. Como listas, os dicionários são mutáveis.

key4	*value4*
key1	*value1*
key3	*value3*
key2	*value2*

Fonte: Model, 2009.

```
>>>
dict((('A','Adenina'),('T','Timina'),('G','Guanina'),('T','Timin
a')))
>>> {'A': 'Adenina', 'T': 'Timina', 'G': 'Guanina'} # Outra
maneira de fazer a mesma coisa
>>> d = {'A': 'Adenina', 'T': 'Timina', 'G': 'Guanina'}
>>> d['A']
>>> d['A'] = 'Hahaha'
>>> d
```

Métodos aplicados a dicionários

Alguns métodos aplicados a dicionários:

```
>>> d.keys()
>>> d.values()
>>> d.items()
```

Exercício 5 - Desafio em grupo (5 alunos, 20 minutos): construir um dicionário para fazer a tradução de RNAs nas sequências de proteínas correspondentes. São

fornecidos aos alunos uma tabela “codons → aminoácidos”, uma lista de expressões e métodos aplicados a dicionários e uma lista com codons que devem ser traduzidos.

Condições

Voltando à lógica booleana (True e False), as operações lógicas em Python podem formar expressões condicionais. Elas permitem a tomada de decisões dependentes das respostas de operações lógicas.

If, else, if - else

Aninhando objetos, condições e estruturas de repetição

Qualquer combinação e em qualquer profundidade:

- Listas podem conter outras listas
- Listas podem conter dicionários

```
>>> M = [[1,2,3],[4,5,6],[7,8,9]] # Matrix 3 x 3 representada em uma lista de listas
>>> M
>>> M[1]
>>> M[1][2]
```

Segundo Dia - Resolução de Problemas

Contextualização do problema 1

Um pesquisador estuda os principais eventos responsáveis pelo reparo do tecido muscular esquelético, a miogênese (formação de fibras musculares) e a fibrogênese (formação de fibrose). Depois de um sonho que teve, decidiu focar em algumas moléculas as quais passou a acreditar serem chaves para o controle desses processos, Myf6*, Myf5** e GDF15***. Ele acredita poder desenvolver uma terapia gênica em modelo murino baseada no gene GDF15. Vale lembrar que as manipulações gênicas acarretam em alterações dos níveis de expressão gênico basal (i.e. normal) de um indivíduo. Para isso ele precisa estudar a expressão normal desses genes. Ele realizou um RNA-Seq para amostras de três tecidos de camundongos (cérebro, fígado e músculo esquelético). O dado de saída é composto de longas tabelas sob as quais ele precisa encontrar algumas informações específicas relevantes para seus experimentos.

1-Ele gostaria de agrupar os genes em 2 classes: Genes relacionados a miogênese e fibrogênese (para isso os dados obtidos devem conter anotações). Quantos genes apareceriam em cada uma das colunas?

2-Encontrando transcritos específicos:
a-Ele gostaria de comparar os dados de expressão (FPKM) dos genes Myf6, Myf5 e GDF15 em cada um dos tecidos. Para isso, uma tabela 3x3 seria visualmente satisfatório.

b-Ele gostaria de fazer uma lista com os 10 genes mais expressos para cada tipo de tecido.

c-Ele gostaria de verificar se os genes Myf6, Myf5 e GDF15 se encontram entre os top 10.

d-Para cada um dos genes (Myf6, Myf5 e GDF15) ele gostaria de gerar uma tabela com os 3 genes mais e 3 genes menos expressos ao redor dos dados de FPKM do músculo esquelético. Para isso, uma tabela 7x3 seria visualmente satisfatório.

e-Como os dados servirão para elaboração de uma terapia gênica e o fígado é um importante órgão metabolizador, qual a posição de Myf6, Myf5 e GDF15 nos 500 genes mais expressos do apenas no fígado? (Prevendo possível hepatotoxicidade).

--

**Também conhecido como MRF4, funciona como regulador positivo da transcrição e está envolvido na maturação dos miotubos durante a diferenciação terminal.*

*** Expresso durante a fase de proliferação de mioblastos. A queda de sua expressão se deve, possivelmente, pelo início da diferenciação.*

****O fator de diferenciação de crescimento-15 (GDF-15) é uma citocina de resposta ao estresse relacionada ao fator transformador de crescimento beta (TGF-ß).*

Problema 1: Manipulação de dados de RNA-seq

Importando bibliotecas usadas

```
import pandas as pd # Importing python library called 'pandas'
import os # Used to check files in directory
import numpy as np
```

Estruturas de dados de pandas

Estruturas Complexas de Pandas (segundo dia)

Serie	
DataFrame	

Series é um objeto unidimensional semelhante a um “array”, consistindo em um “array” de dados e um “array” associado que corresponde aos índices.

Criando o objeto da maneira mais simples:

```
obj = pd.Series([4, 7, -5, 3])
obj
```

Verificando os valores e os índices atribuídos:

```
obj.values
obj.index
```

Acessando e modificando valores:

```
obj[0]
Obj[3]
obj[-1]
```

Criando uma Series com indexação pré-definida e acessar seus valores:

```
obj2 = Series([4, 7, -5, 3], index=['d','b','a','c'])
obj2['a']
```

Exemplos de operações usando esta estrutura de dados incluem filtrar dados ou realizar operações matemáticas, como soma, subtração, exponenciação. Veja que os índices são mantidos:

```
obj2[ obj2 > 0]
obj2 * 2
np.exp(obj2)
```

Series de pandas é muito parecida com um dicionário de python, por conta da estrutura de mapeamento entre valores e chaves. Assim, é possível criar uma Series a partir de um dicionário:

```
sdata = {'Renato':35000,'Franciele':16000,'Leonardo':71000,'Ana':5000}
obj3 = sdata
```

Um exemplo de soma de valores com o mesmo índice em Series diferentes:

```
S1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a','c','d','e'])
S2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a','c','e','f','g'])
print(S1)
print(S2)
print(S1+S2)
```

DataFrame é uma estrutura de dados tabular que contém uma coleção de colunas ordenadas e que podem apresentar tipos de dados diferentes.

A maneira mais comum de criar um DataFrame é usando um dicionário de listas com o mesmo comprimento:

```
data = {'state':['Ohio','Ohio','Ohio','Nevada','Nevada'],
```


```
'year':[2000,2001,2002,2001,2002],
'pop':[1.5,1.7,3.6,2.4,2.9]}
frame = DataFrame(data)
```

Especificando a sequência de colunas no DataFrame criado:

```
DataFrame(data, columns=['year','state','pop'])
```

Dados em colunas podem ser acessados como em Series ou por atributo:

```
frame['state']
frame.state
```

Definindo um novo DataFrame com uma coluna adicional e índices diferentes para as linhas:

```
frame2 = DataFrame(data, columns=['year','state','pop','debt'],
index=['one','two','three','four','five'])
```

Linhas podem ser acessadas de diversas maneiras, incluindo o uso da função ix:

```
frame.ix[2]
frame2.ix['three']
```

Vamos modificar os dados de uma coluna. Por exemplo, a coluna 'debt' inserida acima no frame2 está vazia:

```
frame2['debt'] = 16.5
```

A prática envolvendo dados de RNA-seq

Nesta prática, trabalharemos com resultados tabulados de análises de expressão realizadas com diferentes tecidos do camundongo (*Mus musculus*) C57BL: cérebro, fígado e músculo. C57BL foi o primeiro a ter seu genoma sequenciado e é usado em uma ampla variedade de áreas de pesquisa.

A quantificação de níveis de transcritos foi calculada em RPKM (*reads per kilobase of exon model per million mapped reads*).

Listando arquivos no diretório de trabalho:

```
print(os.getcwd())
print(os.listdir())
print(os.listdir('.')) # Python 2.7
```

Leitura e escrita de arquivos

pandas apresenta funções para a leitura de arquivos com dados tabulados como um objeto DataFrame.

Função	Descrição
`read_csv`	Carrega arquivos separados por vírgula
`read_table`	Carrega arquivos separados por TAB ('\t')
`read_fwf`	Lê dados em arquivos com colunas de tamanho fixo
`read_clipboard`	Versão do `read_table` que lê do clipboard.
`read_excel`	

Lendo os dados de expressão no arquivo de texto com função do pandas:

```
print(pd.read_table('nmeth.1226-S3.txt'))
```

Importando as tabelas com os dados de expressão de sistema nervoso central, fígado e músculo:

```
tabelaExpressaoMouseBrain = pd.read_table('nmeth.1226-S3.txt')
tabelaExpressaoMouseLiver = pd.read_table('nmeth.1226-S4.txt')
tabelaExpressaoMouseMuscle = pd.read_table('nmeth.1226-S5.txt')
```

A anotação dos genes foi obtida da NCBI - incluir a informação de anotação facilita a interpretação dos dados ao analisar resultados. Importando a anotação de genes de camundongos:


```
tabelaAnotacaoCamundongo = pd.read_table('gene_result.txt')
```

Olhando para as primeiras linhas do DataFrame criado. A função **`range()`** em python permite criar uma progressão de números - de 0 a 5 retorna números de 0 a 4:

```
tabelaAnotacaoCamundongo.ix[range(0,5)]
```

Lembre-se: se estivermos interessado em uma coluna específica, basta aninhar as funções:

```
tabelaAnotacaoCamundongo.ix[range(0,5)]['description']
tabelaAnotacaoCamundongo.ix[range(0,5)].description
```

Imprimindo o DataFrame criado com dados de expressão

```
print(tabelaExpressaoMouseBrain)
```

Imprimindo as dimensões do DataFrame:

```
print(tabelaExpressaoMouseBrain.shape) # Printing a tuple representing
the dimensionality of the DataFrame.
print(tabelaExpressaoMouseBrain.size) # Printing the number of
elements in the NDFrame
```

Para imprimir colunas do DataFrame:

```
print(tabelaExpressaoMouseBrain.columns)
```

Olhando para dados da coluna com genes:

```
print(tabelaExpressaoMouseBrain.gene)
print(tabelaExpressaoMouseBrain['gene'])
```

Obtendo informação para a linha com índice ‘1’:

```
print(tabelaExpressaoMouseBrain.ix[1])
```

Recuperando informações de duas linhas (por exemplo, índices 1 e 2):

```
print(tabelaExpressaoMouseBrain.ix[[1, 2]])
print(tabelaExpressaoMouseBrain.ix[[40, 401]])
```

Vamos agora incluir informações de anotação obtida do NCBI nas tabelas com dados de expressão. Precisamos identificar uma coluna em comum entre as diferentes fontes de informação (‘GeneID’ e ‘gid’).

```
tabelaExpressaoMouseBrainAnotacao =
pd.DataFrame(tabelaExpressaoMouseBrain,
columns=['gid','RNAkb','gene','firstRPKM','expandedRPKM','finalRPKM','
fractionMulti','Annotation'])
tabelaExpressaoMouseBrainAnotacao.ix[range(0,5)]
```

Veja que uma coluna não apresenta dados (‘Annotation’). A coluna “Annotation” foi criada com o tipo de dado ‘float’; no entanto, os dados que serão preenchidos são descrições, cadeias de caracteres e não números. Para alterar o tipo de dado para cadeias de caracteres (string):

```
tabelaAnotacaoCamundongo.GeneID =
tabelaAnotacaoCamundongo.GeneID.astype(str)
```

```
for i in range(0,(tabelaExpressaoMouseBrainAnotacao.shape[0]-1)):
 identificador_gene = tabelaExpressaoMouseBrainAnotacao.ix[i].gid
 tabelaExpressaoMouseBrainAnotacao.at[i,'Annotation'] =
tabelaAnotacaoCamundongo.description[tabelaAnotacaoCamundongo.GeneID
== identificador_gene].to_string(index=False)
```

Deletando uma coluna:

```
del tabelaExpressaoMouseBrain['gid']
tabelaExpressaoMouseBrain.ix[range(0,5)]
```

Dimensões do DataFrame podem ser analisados:

```
print(tabelaExpressaoMouseBrain.shape)
```

Tamanho total pode ser verificado usando:

```
print(tabelaExpressaoMouseBrain.size)
```

Transposição:

```
print(tabelaExpressaoMouseBrain.T)
```

Imprimindo apenas os valores do DataFrame:

```
print(tabelaExpressaoMouseBrain.values)
```

Filtrando o DataFrame baseado no nome da coluna (“gene”). O gene *Myf6* (ou *Mrf4*) codifica um fator de transcrição miogênico expresso especificamente em músculo, mas silenciado em fígado e cérebro.

```
print("Brain:
",tabelaExpressaoMouseLiver[tabelaExpressaoMouseLiver['gene'].isin(['M
yf6','Myf5'])])
print("Muscle:
",tabelaExpressaoMouseMuscle[tabelaExpressaoMouseMuscle['gene'].isin([
'Myf6','Myf5'])])
print("Liver:
",tabelaExpressaoMouseBrain[tabelaExpressaoMouseBrain['gene'].isin(['M
yf6','Myf5'])])
```

/# df['A'].isin([3, 6])

Contextualização do problema 2

Um pesquisador está estudando o perfil proteômico da glândula produtora de seda de uma aranha. Para identificar as proteínas, foram realizados experimentos de espectrometria de massas juntamente com auxílio da bioinformática proteômica. Um software foi utilizado para trazer as identificações (nome das proteínas) juntamente com outras informações, p ex. *Accession Number* (NCBI), massa molecular, taxonomia e sequência de fragmentos peptídeos correspondentes.

Problema 2-1: Para cada proteína, seus fragmentos peptídicos correspondentes foram obtidos experimentalmente. O programa usado gerou múltiplas tabelas (uma para cada proteína). Este pesquisador gostaria de ter todos os resultados em uma única tabela mostrando cada uma de suas proteínas identificadas e seus peptídeos correspondentes.
É muito demorado procurar os fragmentos peptídicos correspondentes um a um para cada proteína. Como o Python pode nos ajudar?
Ex:
Sequência da proteína - mioglobina

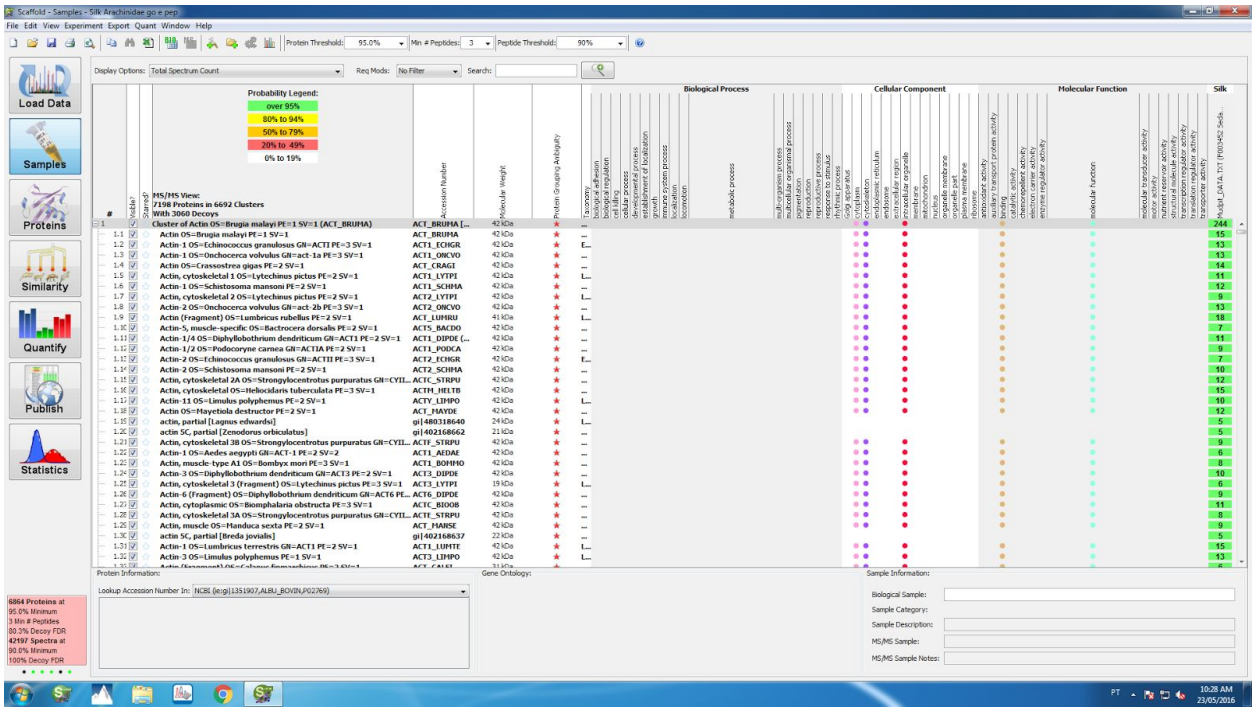
10	20	30	40	50
MGLSDGEWQL	VLNVWGKVEA	DIPGHGQEVL	IRLFKGHPET	LEKFDKFKHL
60	70	80	90	100
KSEDEMKASE	DLKKHGATVL	TALGGILKKK	GHHEAEIKPL	AQSHATKHKI
110	120	130	140	150
PVKYLEFISE	CIIQVLQSKH	PGDFGADAQG	AMNKALELFR	KDMASNYKEL

Sequência dos peptídeos obtidos experimentalmente correspondente a proteína – mioglobina
DGEWQLVLNVWGKVEA
IRLFKGHPETL
EDEMKASE
AMNKALELFRKDMASNYKEL

#	Nome	Número de Acesso	Massa Molecular	Taxonomia	Sequencias de peptídeos
1	Myoglobin	MYG_HUMAN	17 kDa	*Homo sapiens*	DGEWQLVLNVWGKVEA; IRLFKGHPETL; EDEMKASE; AMNKALELFRKDMASNYKEL
2					
3					

Utilizando o mesmo software o pesquisador realizou a anotação funcional das proteínas identificadas através da base de dados *Gene Ontology* (G.O.), a qual proveu função molecular, processo biológico e componente celular destas proteínas.

Problema 2-2: As informações de anotação funcional das proteínas identificadas aparecem descondensadas em subcolunas. É preciso compilar (concatenar) essas informações, de forma que reste apenas 3 colunas principais: processo biológico, componente celular, e função molecular.
Como o Python pode nos ajudar?
Ex.



As sub colunas referentes a:- Biological Process: Biological Adhesion, Biological regulation, cell killing, cellular process, developmental process, establishment of localization, growth, immune system process, localization. Locomotion, metabolic process, multi-organism process, multicellular organismal process, response to stimulus, rhytmic process, antioxidant activity, auxiliary transport protein activity, binding, catalytic activity, chemorepellent activity, electron carrier activity, enzyme regulator activity, molecular transducer activity, motor activity, nutrient reservoir activity, structural molecule activity, transcription regulator activity, translation regulator

activity, transport activity.

As sub colunas referentes a: - Cellular component: Golgi apparatus, cytoplasm, cytoskeleton, endplasmic reticulum, endosome, extracellular region, intracellular organelle, membrane, mitochondrion, nucleus, organelle membrane, organelle part, plasma membrane, ribosome.

Para função molecular apenas existe uma subcoluna - Molecular Function.

Ex.:

#	Nome	Número de Acesso	Massa Molecular	Taxonomia	Sequência de peptídeos	Processo Biológico	Componente celular	Função Molecular
1	Myoglobin	MYG_HUMAM	17kDa	Homo Sapiens	DGEWQLVL NVWGKVEA IRLFKGHPE TL EDEMKASE AMNKALEL FRKDMASN YKEL	Oxygen Trasnsport	-	Oxygen binding

Problema 2-1

Vamos concatenar as várias tabelas com identificação de peptídeos por proteína em uma única tabela.

Começaremos importando as bibliotecas que serão usadas nesta prática:

```
import os # para fazer operações envolvendo o OS do usuário
import pandas as pd # nossos amigos pandas <3
```

Vamos começar listando (e contando) todos os arquivos que queremos juntar. Neste exemplo, eles estão na pasta “proteins”

```
files = os.listdir('proteins')
print('The folder "proteins" has %d files' % len(files))
```

Apesar da extensão “xls”, estes arquivos não são tabelas excel, mas simples tabelas em formato texto separados por tabulações (a.k.a. “tsv”).

Podemos usar a função do pandas “read_table” para criar um dataframe a partir desse tipo de arquivo. Vamos também incluir uma nova coluna na tabela indicando o nome do arquivo de

onde ela veio. Assim, quando concatenarmos as tabelas, poderemos saber de onde veio cada arquivo.

```
def parse_protein_table(fname):
    df = pd.read_table('proteins/{}'.format(fname))
    df['file_name'] = fname
    return df
```

Vamos vamos criar um dataframe com o primeiro da lista “files”

```
df = parse_protein_table(files[0])
```

Agora vamos concatenar as demais tabelas à “df” usando a função “concat” do pandas.

```
for f in files[1:]:
    new_df = parse_protein_table(f)
    df = pd.concat([df, new_df])
```

Finalmente, vamos salvar o dataframe em um arquivo

```
df.to_csv('all_proteins.tsv', sep='\t', index=None)
```

Problema 2-2

Começamos importando as bibliotecas.

```
import pandas as pd
from itertools import chain
import re
```

Criar um DataFrame da tabela de anotações.

```
df = pd.read_excel('proteínas glândula aranha.xltx', skiprows=[0,1,2],
skip_footer=1)
```

Criar um dicionário com o texto das subcategorias de cada uma das três categorias de GO.

```
texto_categorias = {}
texto_categorias[‘biological process’] = """biological adhesion,
biological regulation, cell killing, cellular process, developmental
process, establishment of localization, growth, immune system process,
localization, locomotion, metabolic process, multi-organism process,
multicellular organismal process, response to stimulus, rhythmic
process, antioxidant activity, auxiliary transport protein activity,
binding, catalytic activity, chemorepellent activity, electron carrier
activity, enzyme regulator activity, molecular transducer activity,
motor activity, nutrient reservoir activity, structural molecule
activity, transcription regulator activity, translation regulator
activity, transporter activity, chaperone regulator activity,
chemoattractant activity, metallochaperone activity, pigmentation,
protein tag, reproduction, reproductive process, transporter activity,
viral reproduction"""

texto_categorias['cellular component'] = """Golgi apparatus,
cytoplasm, cytoskeleton, endoplasmic reticulum, endosome,
extracellular region, intracellular organelle, membrane,
mitochondrion, nucleus, organelle membrane, organelle part, plasma
membrane, ribosome"""

texto_categorias['molecular function'] = 'molecular function'
```

Agora vamos usar esse dicionário para criar um dicionário de listas com as categorias.

```
categories = {}
For cat, texto in texto_categorias.items():
    categories[cat] = re.split(r’,\s*’, texto)
```

Vamos salvar as outras colunas não relacionadas a termos GO.

```
colunas_subcategorias = list(chain(*categories.values()))
outras_colunas = [ ]
For col in df.columns:
    If col not in colunas_subcategorias:
        outras_colunas.append(col)
```

Função para concatenar os itens não repetidos de uma lista


```
def concatenar(lista, sep=';'):
    return sep.join(x for x in set(lista) if x != '')

df_semNA = df.fillna('')
```

Criar uma nova tabela, começando com as colunas não relacionadas aos GO.

```
df_concat = df_semNA[outras_colunas].copy()
```

Agora, para cada categoria GO, criar uma coluna dos termos concatenados.

```
for cat in categories:
        df_concat[cat]  =  df_semNA[categories[cat]].apply(concatenar,
axis=1)
```

Finalmente, salvar o arquivo como uma tabela excel.

```
df_concat.to_excel('tabela_concatenada.xlsx', index=None)
```

Glossário computacional

Ambiente integrado de desenvolvimento - ambiente que integra editor e a linguagem de programação, facilitando o trabalho do programador.

Biblioteca - coleção de módulos com definições relacionadas

Coleção (Python) - coleções são objetos que agrupam múltiplos objetos. Há coleções de diferentes tipos, como sequências e “mappings”.

Expressão - a combinação de operador(es) e operando(s)

“Mapping” - coleções cujos elementos são acessados usando chaves.
Mutabilidade - objetos em Python podem ser classificados como mutáveis e imutáveis. Os mutáveis podem ser alterados. String é um exemplo de objeto imutável e lista é mutável.

Operador - símbolo que indica o cálculo/ operação envolvendo um ou mais operandos

Referência (em variáveis da linguagem Python) -

Sequência - coleção de valores lineares, portanto, indexáveis (termo ambíguo).

Tipagem estática - em linguagens deste tipo, nomes de variáveis estão associadas a um objeto e a um tipo.

Tipagem dinâmica - em linguagens deste tipo, nomes de variáveis estão associadas a um objeto e o tipo de variável é verificado durante o tempo de execução.

Referências e páginas relevantes

Artigos

Mortazavi, Ali, et al. "Mapping and quantifying mammalian transcriptomes by RNA-Seq." *Nature methods* 5.7 (2008): 621-628.

Ekmekci, Berk, Charles E. McAnany, and Cameron Mura. "An introduction to programming for bioscientists: A python-based primer." *PLoS computational biology* 12.6 (2016): e1004867.

Bassi, Sebastian. "A primer on python for life science researchers." *PLoS Computational Biology* 3.11 (2007): e199.

Livros

McKinney, Wes. Python for data analysis: Data wrangling with pandas, NumPy, and IPython. " O'Reilly Media, Inc.", 2012.

Menezes, Nilo Ney Coutinho. *Introdução à programação com Python–2ª edição: Algoritmos e lógica de programação para iniciantes*. Novatec Editora, 2016.

Lutz, Mark. Learning Python: Powerful Object-Oriented Programming. " O'Reilly Media, Inc.", 2013.

Lutz, Mark. Python Pocket Reference: Python In Your Pocket. " O'Reilly Media, Inc.", 2014.

Buffalo, Vince. *Bioinformatics data skills: Reproducible and robust research with open source tools*. " O'Reilly Media, Inc.", 2015.

Matthes, Eric. Curso Intensivo de Python: Uma introdução prática e baseada em projetos à programação. " O'Reilly Media, Inc.", 2016.

Ramalho, Luciano. Python Fluente: Programação clara, concisa e eficaz " O'Reilly Media, Inc.", 2015.

Model, Mitchell L. *Bioinformatics Programming Using Python: Practical Programming for Biological Data*. " O'Reilly Media, Inc.", 2009.

Páginas interessantes

Documentação da linguagem Python 3.
https://docs.python.org/3/

Rosalind - uma plataforma para aprender bioinformática e programação
http://rosalind.info/problems/locations/

Software Carpentry
http://software-carpentry.org/