Prof. Raquel C. de Melo-Minardi, Ph.D.
Departamento de Ciência da Computação / UFMG
10 de agosto de 2020

Nesse módulo, apresentaremos um curso de introdução a programação utilizando para os exemplos práticos a linguagem de programação Python.

Por que escolher Python para Bioinformática?

Python é uma linguagem de programação criada em 1991 por Guido van Rossum com o objetivo de facilitar as implementações em detrimento do esforço computacional. Em outras palavras, ela preza a legibilidade e o minimalismo do código fonte sobre a velocidade de execução. Roda em diversas plataformas e pode ser categorizada como de propósito geral, de alto nível, imperativa, funcional e procedural, além de interpretada, de tipagem dinâmica e forte e, por fim, orientada por objetos. Tem sido amplamente utilizada para processamento de textos, dados científicos e criação de CGIs para páginas dinâmicas para a web.

A terceira versão da linguagem foi lançada em dezembro de 2008, é denominada Python 3 e não é compatível com a família Python 2. Ela foi criada, segundo seus desenvolvedores, para corrigir falhas que foram descobertas no padrão pré-existente e para limpar os excessos das versões anteriores. Nesse curso, focaremos na programação em Python3, cuja sintaxe é incompatível com a de Python 2.

Por que a linguagem Python foi escolhida para esse curso? Há uma grande variedade de linguagens de programação disponíveis. Está além do escopo desse curso uma definição mais completa das principais características das linguagens de programação modernas mas é importante que o estudante saiba que a escolha de uma linguagem não deve ser apenas subjetiva ou por uma preferência pessoal ou conhecimento prévio da linguagem. O que quero dizer é que tenho conhecido, ao longo da minha carreira de professora, estudantes que não se sentem motivados a aprenderem novas linguagens mas utilizam sempre a linguagem que já domina independente do projeto que estejam desenvolvendo. Essa não é uma boa idéia. A escolha de uma linguagem adequada pode simplificar muito o desenvolvimento de seu projeto.

Comecei meus estudos em bioinformática em 2003 e, naquela época, eu era fluente em C e Java, além de outras linguagens para desenvolvimento de aplicações com interfaces gráficas ou para a web. Na minha primeira tarefa de programação para bioinformática, deparei-me com inúmeras dificuldades no uso da linguagem C para a extração das informações que necessitava de um arquivo de coordenadas estruturais de proteínas (do PDB, *Proteína Data Bank*). Foi natural começar a resolver o problema em C, que era a linguagem que eu mais tinha utilizado nos 4 anos precedentes durante o bacharelado em ciência da computação. Contudo, não gastei mais que 30 minutos para perceber que eu precisava de uma solução mais adequada para aquele problema e que essa solução envolvia a escolha de outra linguagem.

Foi então que descobri o Perl e me pareceu perfeitamente adequado por conter alguns recursos que o C não apresentava. O que me chamou mais a atenção no Perl, e a meu ver o torna adequado para a bioinformática, é a sua capacidade de detecção de padrões e motivos em dados e pela forma de trabalhar com as expressões regulares, das quais trataremos oportunamente nesse módulo. No meu ponto de vista, a grande vantagem da linguagem Perl para a bioinformática é a sua flexibilidade que nos permite obter e tratar os tipos de dados que a bioinformática nos apresenta. Há quem goste da linguagem Perl pois ela é considerada uma

das linguagens padrão de bioinformática sendo disponíveis inúmeros repositórios e bibliotecas como o BioPerl, por exemplo.

Alguns anos mais tarde, notei o crescente uso do Python pelos estudantes de bioinformática e de ciência da computação. Passei a observar o seu crescimento e decidi utilizá-lo em minha disciplina de pós-graduação em 2018 tendo notado que seu uso ultrapassava o de Perl e ela compartilha a maioria das qualidades positivas do Perl. Existe também uma biblioteca análoga ao BioPerl, a BioPython.

Assim como Perl, Python compartilha a principal desvantagem de Perl que é o baixo desempenho por ser interpretada e tirada dinamicamente. Em contrapartida, Python não apresenta as seguintes desvantagens de Perl:

- induzir maus hábitos de programação para novatos
- existir diversas construções para se expressar a mesma tarefa.

Por fim, após aprender Python, duas outras linguagens bastante proeminentes em bioinformática e que devem ser aprendidas pelos estudantes: Perl e R. Perl é uma competidora direta de Python. R tem um propósito um pouco diferente e um grande foco em análises estatísticas, nicho em que é a melhor e não há competidoras. Não pretendemos nos estender nessa comparação entre as linguagens até porque acreditamos que as três deveriam ser aprendidas por um bioinformata. Nossa proposta nesse curso é começar por Python.

Apenas por curiosidade, há quase uma década, [Mathieu e Gillings, 2008] compararam o uso de memória e o tempo de execução de três algoritmos tradicionais em bioinformática (Algoritmo de Seller para casamento de strings, algoritmo para *Neighbor-Joining* e um parser para pós-processamento do BLAST), implementados em programas usando seis diferentes linguagens de programação (C, C++, C#, Java, Perl e Python). As implementações em C e C++ foram mais rápidas e usaram menos memória. Essas linguagens geralmente utilizaram mais linhas de código. Java e C# apresentam um compromisso, ou ficam no meio termo, entre a flexibilidade de Perl e Python e o bom desempenho de C e C++.

Uma colocação final antes de começarmos a programar é que a linguagem é apenas uma ferramenta. A **lógica de programação** sim é a competência essencial que um novato em bioinformática precisa desenvolver. Um bom programador, com um pequeno esforço e curto intervalo de tempo, aprende novas linguagens com reconhecida facilidade.

PYTHON

O que é o Python?

Python é uma linguagem de programação de **propósito geral** originalmente desenvolvida para manipulação de texto e agora usada para uma ampla gama de tarefas, incluindo administração de sistemas, desenvolvimento web, programação de rede, desenvolvimento de GUI e muito mais [Site Python Documentation].

Ela foi concebida para se fácil de usar, pequena, elegante, mínima. Suas principais características são a facilidade de uso, suporte à programação processual e orientação por objetos (OO). Possui poderoso suporte ao processamento de texto.

Em geral, um programa em Python se parece com um programa em C ou em outras linguagens de propósito geral: existem variáveis, expressões, atribuições, blocos de código, estruturas de controle e rotinas. Python possui variáveis simples, de tipos básicos, além de variáveis compostas como tuplas, conjuntos, listas e dicionários. Elas também possuem mecanismos para tratamento de expressões regulares. Isso simplifica enormemente o tratamento de texto em geral. Suporta ainda o tratamento de estruturas de dados complexas e a orientação por objetos.

Uma característica final e muito digna de nota é que Python possui gerenciamento de memória automático e **tipagem dinâmica** que é uma característica de determinadas linguagens de programação, que não exigem declarações de tipos de dados e são capazes de escolher que tipo utilizar dinamicamente para cada variável, podendo alterá-lo durante a compilação ou a execução do programa. A grande maioria das linguagens de programação (como C, C++ e Java) usam a **tipagem estática**, que exige a declaração de quais dados poderão ser associados a cada variável antes de sua utilização.

A tipagem dinâmica nos dá grande flexibilidade e facilidade de programação em Python, fazendo muitas tarefas árduas serem bastante simplificadas quando implementadas em Python. Contudo, como toda flexibilização, isso também nos traz uma maior responsabilidade. O objetivo principal dos sistemas de tipos em linguagens de programação é reduzir erros em programas de computador. Esses erros são reduzidos à medida que se define previamente interfaces entre diferentes partes de um programa, normalmente através dos tipos e, em seguida, é possível verificar que as partes tenham sido conectadas de maneira consistente. É como se fosse estabelecido um contrato entre as diferentes partes do software e os termos desse contrato pudessem ser verificados. Essa verificação pode acontecer estaticamente (em tempo de compilação), dinamicamente (em tempo de execução) ou como uma combinação destes.

Sumário

Vantagens de Python:
- É portátil sendo possível de ser utilizada em diversas plataformas
- É simples, concisa e fácil de ler e aprender
- É excelente para manipulação de texto
- É uma linguagem de alto nível
- Foi desenvolvida com o ideal de software livre

Desvantagens de Python:
- Devido a ser interpretado, é lento
- Os scripts permitem leitura, impedindo de esconder o código fonte

EXECUTANDO UM PROGRAMA EM PYTHON

Para executar um programa ou script em Python 3 de uma linha de comando UNIX você deve usar o seguinte comando:

```
>python3 programa.py
```

Onde “programa.py” é o nome do arquivo texto com o código fonte do seu programa.

No Linux, de forma alternativa, você também pode colocar a seguinte linha como primeira linha de seu script

```
#!/usr/bin/python
```

E rodar o script diretamente após dar-lhe permissão de execução. No linux, isso pode ser feito através do seguinte comando:

```
>chmod 755 programa.py
```

PRIMEIRO PROGRAMA EM PYTHON

Todo primeiro programa que se aprende em uma linguagem é o famoso “Hello world” e a única coisa que faz é imprimir algo na tela. Pois bem, para testarmos nosso programa em Python, abra um editor de texto puro de sua preferência (vi, vim, sublime, atom, notepad++, kate, etc.) e digite o seguinte comando:

```
print('Hello world!!!')
```

Execute então seu programa e verifique se ele funcionou.

ALGUNS COMENTÁRIOS SOBRE A SINTAXE BÁSICA

Um programa Python é uma sequência de uma ou mais sentenças. Não é preciso uma função principal como, por exemplo, em C que é preciso que o programa principal esteja dentro de uma rotina “main".

Comentários começam com o símbolo de jogo da velha:

```
# Isso é um comentário em Python e será ignorado pelo interpretador
```

Espaços e quebras de linha são relevantes e geram erros de interpretação se não usados corretamente.

```
print(
                'Hello world!!!'
        )
```

Tanto aspas simples quanto aspas duplas podem ser utilizadas em Python para delimitar *strings*, ou cadeias de caracteres:

```
print("Hello world!!!")
print('Hello world!!!')
```

Números não precisam ser colocados entre aspas:

```
print(12)
```

Alguns conceitos ficarão mais claros ao longo do curso quando introduzirmos o conceito e uso de variáveis ou mesmo de funções.

VARIÁVEIS

Segundo [Celes et al., 2004]

Variáveis representam um espaço de memória do computador para armazenar determinado tipo de dado.

Em linguagens estaticamente tipadas, como C, todas as variáveis precisam ser explicitamente declaradas antes de seu uso. A **declaração** consiste em especificar o **nome** e **tipo**. O nome serve para associar o dado armazenado ao espaço de memória da variável e o tipo determina a natureza do dado que será armazenado. Nessa categoria de linguagem, uma variável só pode receber valores do tipo de dados previamente acordado na declaração da mesma. Por exemplo, se declaramos uma variável do tipo **inteiro**, só podemos armazenar números inteiros nessa variável. Em C, existem alguns tipos básicos que são *char* (caracter), *short* (inteiro pequeno), *int* (inteiro), *long* (inteiro grande). Esses tipos diferem entre si pelo espaço de memória que ocupam e, consequentemente, pelo intervalo de valores que conseguem representar. Por exemplo, um número *char* é representado em 1Byte (8 bits) e pode representar $2^8 = 256$ valores distintos. Um *short* é representado por 2 Bytes e *long* por 4 Bytes.

Em Python, a primeira atribuição de valor gera uma declaração de variável, definindo momentaneamente o seu tipo como o tipo do dado atribuído. Nomes de variáveis podem surgir ao longo do código e podem assumir dinamicamente tipos primitivos *int, str* (cadeia de caracteres), *bool* (valores Booleanos ou lógicos), *list*, (lista), *tuple* (tupla) e *dic* (dicionário). Por um lado, facilita enormemente a vida do programador não ter que se preocupar em declarar uma variável sabendo o tipo e o volume dos dados quando está programando, por outro lado, a menor de possibilidade de validação dos dados pode gerar inúmeros problemas em tempo de execução devido à dados ausentes, mal coletados entre diversos problemas difíceis de prever. Mais uma vez, destacamos que o excesso de flexibilidade e indulgência da linguagem traz uma grande responsabilidade ao programador que pode cometer erros inadvertidamente. Em Perl, esses problemas seriam ainda piores.
Essas definições de tipos explicadas até o momento se tratam de tipagem do dados. Há outra categorização de tipos de variáveis mas em termos da organização da estrutura de dados envolvida. Vamos definir então esses tipos:

- **Variáveis simples** ou atômicas: são variáveis associadas a dados simples compostos por apenas um elemento como um número ou um caracter. Exemplos: um caracter, um aminoácido.
- **Variáveis compostas**: são variáveis definidas por um nome associado a diversos elementos. Exemplos: um texto, a sequência de aminoácidos de uma proteína.
  - **Variáveis homogêneas**: são variáveis compostas por elementos de um mesmo tipo de dados. Exemplo: uma sequência de aminoácidos é uma variável composta de inúmeros caracteres, ou seja, elementos do mesmo tipo.

- **Variáveis heterogêneas**: são variáveis que armazenam conjuntos de dados formados por tipos de dados diferentes (campos do registro) em uma mesma estrutura. Exemplo: uma variável proteína que guarde dentro de sua estrutura os campos id (*str*), tamanho (*int*), massa (*float*), sequência (*str*).

Como tudo em Python está relacionado a sua enorme flexibilidade, as variáveis em Python também não seguem essas definições de forma rígida. Python possui variáveis simples e compostas mas não há nenhuma forma de garantir que uma variável composta seja homogênea, ou seja, a princípio, todas as variáveis compostas em Python são heterogêneas. Vamos descrever em detalhes a seguir os tipos de variáveis em Python.

Tipos de variáveis

Python tem diversos tipos de variáveis:

Simples

Uma variável simples armazena um único valor, sejam caracteres, inteiros, complexos, booleanos ou números de ponto flutuante.

Compostas

Uma variável composta pode armazenar diversos de valores:

```
proteinas = ['1A6M', '2MM1', '1CHO', '2PTI', '1A6N']
tamanhos = [153, 155, 233, 245, 150]
heterogeneo = ['1A6M', 153, '2MM1', 155, '1CHO', 233, '1A6N', 245]
```

Como exemplificado na terceira linha, as variáveis compostas em Python podem acomodar, ao mesmo tempo, diversos tipos de dados. Cabe destacar que, em Python, sempre iniciam a indexação por “0”. Veja alguns exemplos de como as variáveis compostas podem ser acessados diretamente:

```
print(proteina[0]) # Imprime 'ALA'
print(proteina[1]) # Imprime 'CYS'
print(tamanhos[0])  # Imprime 153
```

Uma coisa interessante de se chamar a atenção é de que hora usamos a notação “proteinas” e hora “proteinas[0]”. A notação sem os colchetes é usada quando denotamos uma variável composta. Quando acessamos posições individuais da variável, dizemos que estamos no contexto simples e o acessamos através de um *offset* (deslocamento) dentro de colchetes.

Figura 1 - Uma **variável simples** poderia ser comparada a uma residência na qual podemos hospedar apenas uma família

Figura 2 - Uma **variável composta** pode ser comparada a uma rua na qual há diversas residências. Cada residência tem sua numeração e essa numeração é contígua e inicia do índice “0”

Com relação às variáveis compostas, Python apresenta:

- **sequências**: objetos ordenados e finitos
- **dicionários**: conjuntos de elementos indexados por chaves (que podem ser sequências de caracteres)

Sequências

As sequências podem ser de dois tipos:

- **sequências imutáveis**: objetos ordenados e finitos
    - ***strings***: cadeias de caracteres
    - ***tuples***: dois ou mais elementos de qualquer tipo dentro de parênteses e separados por vírgula
- **sequências mutáveis**:
    - ***lists***: conhecidas em outras linguagens como vetores ou arranjos
    - ***sets***: coleções não ordenadas e não indexadas escritas entre chaves
    - ***dictionaries***: conjuntos de elementos indexados por chaves (que podem ser sequências de caracteres)

Strings

Em Python, uma variável do tipo *string* armazena uma cadeia de caracteres. Veja um exemplo da sintaxe da declaração e inicialização de uma variável do tipo *string*:

```
sequencia = 'ACTTGGCAGTGACAAAGTGCATGGGGGACT'
```

Como todos os outros tipos de dados, *strings* são objetos em Python. Isso implica na existência de diversos métodos para se manipular strings embutidos no próprio objeto.

Para descobrir quais são os métodos de um objeto você pode digitar o nome da variável na interface interativa do Python, seguido de um “.” e “tab”. Ele listará as possibilidades de métodos implementados para a classe.

Apresentaremos a seguir alguns dos métodos mais usados de *strings*.

O método “replace” visa a substituição de partes (trechos) em *strings* recebendo como argumentos:

```
sequencia = sequencia.replace('AAA', 'CCC')
```

1. a subsequência a ser encontrada e substituída: primeiro argumento
2. a subsequência que substituirá a subsequência buscada: segundo argumento

conforme a sintaxe abaixo:

O método “count” conta o número de ocorrências de um trecho ou subsequência em uma *string*:

```
sequencia.count('AAA')
```

O método "find" recebe como argumento uma sequência e retorna em que posição se encontra essa subsequência :

```
sequencia.find('AAA')
sequencia[sequencia.find('AAA')]
```

O método "split" quebra e separa uma *string* por um certo trecho, retornando uma lista:

```
sequencia.split('A')
```

E e o trecho for passado em branco, retorna uma lista com um elemento apenas que é a sequência original.

O método “join" pode ser usado para unir *strings* conforme o exemplo abaixo:

```
'AAA'.join(sequencia)
```

Há ainda diversas funções que nos permitem mudar os caracteres para **maiúsculo** / **minúsculo** como nos exemplos a seguir:

```
sequencia.upper()
sequencia.lower()
sequencia.lower().capitalize() # Apenas a primeira letra em maiúsculo
sequencia.title() # Cada primeira letra de palavra em maiúsculo
sequencia.swapcase() # Inversão de maiúsculo para minúsculo e vice-versa
```

Existem ainda as funções "isupper", "islower", "istitle", "isalnum", "isalpha", "isdigit", "isspace" que retornam valores booleanos de acordo com os testes de capitalização realizados nas *strings*.

É possível justificar uma *string* em um determinado espaço usando os métodos “ljust”, “rjust” e “center” conforme os exemplos:

```
'AAA'.ljust(15)
'AAA'.rjust(15)
'AAA'.center(15)
```

E ainda remover os espaços no início e fim da *string:*

```
sequencia.strip()
sequencia.lstrip()
sequencia.rstrip()
```

Não é necessário usar um método em particular para obter trechos de uma *string* pois Python nos permite acessar essas partes através de índices como exemplificamos a seguir:

```
sequencia[6:9]
sequencia[:9]
sequencia[6:]
```

Note que “sequencia[6:9]” resultará nos caracteres de 6 a 8 da *string* uma vez que o segundo índice é sempre um posterior, ou seja, trata-se do intervalo fechado-aberto [6,9). Caso um dos índices não seja especificado, o resultado será o índice “0” para o início e o último índice para o final. Pode-se ainda usar índice negativo e isso

significará percorrer de trás para frente. Por exemplo, “sequencia[0:-2]” retorna a sequência do início até o antepenúltimo inclusive.

Finalmente, um método genérico de Python que se aplicará a diversos tipos de variáveis compostas é o “len" que é usado para retornar o comprimento:

```
len(sequencia)
```

Tuplas

Tupla é uma lista imutável. O que diferencia a lista da tupla é que a primeira pode ter elementos adicionados a qualquer momento, enquanto que a segunda estrutura, após definida, não permite a adição ou remoção de elementos. Tuplas são definidas de forma análoga a listas, mas o que as diferencia são os caracteres que as delimitam. Listas tem seus elementos delimitados por colchetes, enquanto que a tuplas tem seus elementos delimitados por **parênteses**.

```
t = ('10', 'segunda-feira', 'Fevereiro', 2022)
```

A ordem dos elementos numa tupla será a ordem na qual estes foram definidos, ou seja, não é possível ordenar em tempo de execução os elementos. O primeiro elemento de uma tupla também possui índice igual a 0 e o último índice igual n-1.

Veja a seguir outras formas de se declarar uma tupla:

```
#tupla declarada sem o uso de parentesis
t1 = 1, 2, 3

#tupla declarada com o uso de parênteses
t2 = (1, 2, 3)

#tupla com um único elemento
t3 = 1,

#tupla vazia
t4 = ()
```

Note que, na realidade, não são os parênteses que definem uma tupla, mas as **vírgulas**.

Apresentaremos agora alguns operadores que são úteis na manipulação de tuplas.

O operador "in" pode ser usado para retornar se um determinado elemento existe (pertence a) em uma tupla:

```
5 in tupla # Retornara True ou False
```

Através dos operadores “+" e “*” é possível **concatenar** tuplas:

```
t = ('a', 'b') + ('c',) # t = (‘a’, ‘b’, ‘c')
t += (3,) # t = (‘a’, ‘b’, ‘c’, 3)

t = t * 3 # t = (‘a’, ‘b’, ‘c’, ‘a’, ‘b’, ‘c’, ‘a’, ‘b’, ‘c’)
```

Note que esses operadores **NÃO** fazem operações aritméticas com as tuplas e sim concatenam tuplas.

Tuplas tem apenas dois métodos que apresentamos a seguir.

O método "index" retorna a primeira ocorrência do elemento em uma tupla:

```
tupla.index('A')
```

E o método “count" que retorna a o número de ocorrências do elemento em uma tupla:

```
tupla.count(‘A'))
```

Da mesma forma como ocorre com as *strings*, não se usa métodos para obter partes das tuplas mas apenas indexação:

```
tupla[6:9]
tupla[:9]
tupla[6:]
```

Índices negativos funcionam de forma análoga para todas as variáveis compostas e já foram discutidos quando introduzidos em *strings*.

Por fim, o método “len" nativo de Python também retorna o tamanho de uma tupla:

```
len(tupla)
```

Listas

Durante o desenvolvimento de software, independentemente de plataforma e linguagem, é comum a necessidade de lidar com listas. Por exemplo, elas podem ser empregadas para armazenar nucleotídeos de uma sequência de DNA ou aminoácidos de uma proteína. Uma lista é uma estrutura de dados composta por itens organizados de forma linear, na qual cada um pode ser acessado a partir de um índice, que representa sua posição na coleção (iniciando em zero).

```
proteinas = ['1A6M', '2MM1', '1CHO', '2PTI', '1A6N']
tamanhos = [153, 155, 233, 245, 150]
heterogeneo = ['1A6M', 153, '2MM1', 155, '1CHO', 233, '1A6N', 245]
```

Como exemplificado na terceira linha, uma lista em Python pode ter diversos tipos de dados, ou seja, é uma variável composta heterogênea.

Em Python, uma lista é representada como uma sequência de objetos separados por vírgula e dentro de colchetes "[]", assim, uma lista vazia, por exemplo, pode ser representada por colchetes sem nenhum conteúdo:

```
proteina = []
proteina = ['ALA', 'LYS', 'GLY', 'GLU', 'ALA']
proteina1 = ['A', 'K', 'G', 'E', 'A']
proteina2 = ['A', 'K', 'G', 'E', 'A']
uniao = [proteina1, proteina2, 'K', 'G', 'E']
```

Listas em Python sempre iniciam a indexação por “0”. Veja alguns exemplos de como elas podem ser acessados diretamente:

```
proteina = ['ALA', 'CYS', 'ASP', 'GLU', 'HIS']
tamanhos = [153, 155, 233, 245, 150]
print(proteina[0]) # Imprime 'ALA'
print(proteina[1]) # Imprime 'CYS'
print(tamanhos[0])  # Imprime 153
```

O tamanho ou o número de elementos de uma lista, ou o número de itens que a compõem, pode ser obtido a partir da função “len":

```
proteina = ['ALA', 'CYS', 'ASP', 'GLU', 'HIS']
print(len(proteina)) # Imprime 5
```

É possível concatenar listas por meio dos operadores de adição "+" e “*”. O código a seguir traz alguns exemplos dessas operações:

```
proteina1 = ['A', 'K', 'G', 'E', 'A']
proteina2 = ['A', 'K', 'G', 'E', 'A']
concatenacao = proteina1 + proteina2
print(concatenacao)
# imprime ['A', 'K', 'G', 'E', 'A', 'A', 'K', 'G', 'E', 'A']
multiplicacao = proteina1 * 2
print(multiplicacao)
# imprime ['A', 'K', 'G', 'E', 'A', 'A', 'K', 'G', 'E', 'A']
```

Utilizamos o operador “in" para verificar a pertinência de elementos à lista. Ele retornará True se objeto pertencer ao conjunto, e False caso contrário.

O Python oferece as funções "min", "max" e "sum", através das quais é possível encontrar, respectivamente, o menor valor, o maior valor ou ainda realizar a soma de todos os elementos de uma lista.

Até então apresentamos operadores e funções que recebem uma lista como argumento, retornam um resultado, mas não efetuam alterações na sua estrutura. A partir de agora apresentaremos métodos pertencentes ao tipo lista.

O método “append" adiciona um novo elemento no final da lista:

```
lista = [4, 5, 3, 2, 7, 8, 1, 0]
lista.append(9)
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Antes do *append*	4	5	3	2	7	8	1	0	
Após o *append(9)*	4	5	3	2	7	8	1	0	**9**

Para inserir itens em uma posição específica, utilizamos o método "insert" que, além do elemento a ser inserido, recebe também o índice que ele deve assumir:

```
lista = [4, 5, 3, 2, 7, 8, 1, 0]
lista.insert(6, 9)
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Antes do *insert*	4	5	3	2	7	8	1	0	
Após o *insert(6, 9)*	4	5	3	2	7	8	**9**	1	0

O método “pop" remove o último item da lista e o retorna como resultado da operação:

```
lista = [4, 5, 3, 2, 7, 8, 1, 0]
lista.pop()
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Antes do *pop*	4	5	3	2	7	8	1	**0**
Após o *pop*	4	5	3	2	7	8	1	

Caso seja necessário remover um índice específico, basta informá-lo como argumento:

```
lista = [4, 5, 3, 2, 7, 8, 1, 0]
lista.pop(3)
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Antes do *pop*	4	5	3	**2**	7	8	1	0
Após o *pop*	4	5	3	7	8	1	0	

Há situações em que o índice de elementos que se deseja remover é desconhecido e desejamos remover o item a partir do seu valor e para tal usamos o método remove(item). Ele removerá o primeiro item com o determinado valor:

```
lista = [4, 5, 3, 2, 7, 8, 1, 0]
lista.remove(2)
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Antes do *remove*	4	5	3	**2**	7	8	1	0
Após o *remove*	4	5	3	7	8	1	0	

Há também funções prontas para inverter e ordenar arranjos:

O método "reverse" inverte um arranjo

```
lista = [4, 5, 3, 2, 7, 8, 1, 0, 11]
lista.reverse()
print(lista)# Imprime 11, 0, 1, 8, 7, 2, 3, 5, 4
```

O método "sort" ordena um arranjo em ordem crescente:

```
lista = [4, 5, 3, 2, 7, 8, 1, 0, 11]
lista.sort()
print(lista)# Imprime 0, 1, 11, 2, 3, 4, 5, 7, 8
```

O método "count" retorna o número de ocorrências de determinado objeto, passado como parâmetro, em uma lista:

```
lista = [4, 5, 3, 2, 7, 8, 1, 0, 11]
print(lista.count(4)) # Imprime 1
```

Cabe por fim uma reflexão de quando usar tuplas e quando usar listas. Você notou que as listas são extremamente mais flexíveis que as tuplas já que são mutáveis enquanto as primeiras não o são. É importante destacar que tuplas são mais rápidas que listas. Se você precisa apenas definir um conjunto de valores constantes e depois percorrê-lo e consultá-lo de alguma forma, opte por tuplas. É como se pensássemos que “tupla congela uma lista e lista descongela uma tupla”.

Conjuntos

Um conjunto é uma coleção não ordenada e não indexada. No Python, os conjuntos são denotados por chaves:

```
nucleotideos = {'A', 'C', 'G', 'T'}
```

Os conjuntos não são ordenados, portanto, você não pode ter certeza da ordem na qual os itens serão exibidos.

Há alguns operadores que podem ser usados para operar em objetos do tipo conjunto. O operador "in" pode ser usado para retornar se um determinado elemento existe (pertence a) em um conjunto:

```
'A' in nucleotideos # Retornará True ou False
```

Depois que um conjunto é criado, você não pode alterar seus itens. Entretanto, você pode adicionar um novo item através do método “add":

```
nucleotideos.add('X')
```

assim como adicionar vários elementos de uma só vez usando o método "update" e passando uma lista como argumento:

```
nucleotideos.update(['A','B','C'])
```

Existem várias maneiras de associar dois ou mais conjuntos no Python. Você pode usar o método “union” que retorna um novo conjunto contendo todos os itens de ambos os conjuntos :

```
nucleotideosM = {'A', 'C', 'G', 'T'}
nucleotideosm = {'a', 'c', 'g', 't'}
nucleotideos = nucleotideosM.union(nucleotideosm)
```

ou o método “update" que insere todos os itens de um conjunto em outro:

```
nucleotideosM = {'A', 'C', 'G', 'T'}
nucleotideosm = {'a', 'c', 'g', 't'}
nucleotideosM.update(nucleotideosm)
```

É importante destacar que itens idênticos duplicados são sempre unificados devido à própria definição do que é um conjunto.

Ainda advindo da definição de conjunto, há inúmeras operações que podemos desejar realizar com conjuntos tais como:

- identificar a diferença entre conjuntos: método "difference"
- verificar se dois conjuntos possuem interseção: método "intersection"
- verificar se dois conjuntos são disjuntos: método "isdisjoint"
- verificar se um conjunto é subconjunto de outro: método "issubset"
- vertical se um conjunto é superconjunto de outro: método "issuperset"

É possível remover elementos de um conjunto usando os métodos "remove", "discard" e “pop”:

```
nucleotideos = {'A', 'C', 'G', 'T'}
nucleotideos.remove('A')
```

A diferença é que o método “remove" gera erro se o elemento não existe no conjunto e o “discard" não levanta esse erro. O método “pop” remove e retorna o último elemento do conjunto e como um conjunto não é ordenado, não se tem como prever que elemento será retornado.

O método “clear" esvazia um conjunto:

```
nucleotideos.clear()
```

Por fim, o método “len” que já foi apresentado anteriormente no contexto das outras variáveis compostas retorna o tamanho de um conjunto:

```
len(conjunto)
```

Apresentamos agora uma síntese comparativa entre algumas tarefas comuns que podemos desejar realizar com as variáveis compostas em Python:

	String	Tupla	Lista	Conjunto
Criar	`s = ''`	`t = ()`	`l = []`	`c = {'t'}`
Obter parte	`s[6:9]`	`t[6:9]`	`l[6:9]`	-
Substituir	`s.replace('AAA','CCC')`	-	-	-
Contar	`s.count('AAA')`	`t.count('AAA')`	`l.count('AAA')`	-
Encontrar	`s.find('AAA')`	`t.index('AAA')`	`l.index('AAA')`	-
Verificar pertencimento	`'AAA' in s`	`'AAA' in t`	`'AAA' in l`	`'AAA' in c`
Quebrar	`s.split('A')`	-	-	-

Dicionários

Uma variável do tipo dicionário é um *hash*. Representam um conjunto não ordenado de pares chave-valor. Apenas por curiosidade, segundo [Ziviani, 2004], *hash* é um verbo no idioma inglês e e significa (i) fazer picadinho (de carne ou vegetais para cozinhar) ou (ii) fazer uma bagunça. Ao contrário do que ocorre com os arranjos, nos quais os índices são números inteiros, crescentes, contíguos e que podem ser usados diretamente como offsets para endereçamento em memória, nos *hashes* as chaves são quaisquer cadeias de caracteres que devem obrigatoriamente ser transformadas em endereços de memória para serem usadas como chave. Esse processo de transformação de uma chave (cadeia de caracteres) consiste em dois processos: (i) computar uma função de transformação, que “pica” a cadeia de caracteres em pedaços e os utiliza para gerar um endereço em uma tabela (memória) (ii) considerando que duas ou mais chaves podem

ser transformadas em um mesmo endereço de uma tabela, implementar um método de tratamento de colisões. Há diversos algoritmos para computar essas funções de transformação e tratar colisões e sua explicação está além do escopo desse curso. O importante é explicar que o Python já os implementa para nós e simplesmente podemos usar a estrutura de dados dicionário (*dict*) de forma transparente a esses métodos.

Em Python, há duas formas para se inicializar um dicionário:

```
dic = {} # Inicializa ou reinicializa um dicionário existente
dic = {'5GJA:A':61, '5GJ4:B':177, '5GJ4:C':61, '5GJK:D':177, '5GP1:B':268}
```

Essas duas possibilidades apresentadas servem para inicializar um dicionário com dados de interesse mas também reinicializam um dicionário que por ventura já possua valores. É importante destacar que, nesse último caso, o dicionário existente será descartado e conterá apenas os novos valores da inicialização.

Uma outra outra possibilidade de inicialização, mas para posições individuais, que tem a seguinte sintaxe:

```
dic['XXX'] = 999
```

Nessa versão, uma nova chave seria criada caso ainda não exista ou o valor seria atualizado caso já exista.

É interessante mencionar que, ao contrário do que ocorre nos listas, dicionários não tem ordem particular em termos de armazenamento em memória.

Itens de dicionários também podem ser acessados diretamente:

```
dic = {}
dic = {'5GJA:A':61, '5GJ4:B':177, '5GJ4:C':61, '5GJK:D':177, ‘5GP1:B':268}
```

Mas é importante saber que o acesso a itens inexistentes retornará um erro. Por essa razão, em alguns casos, é preciso verificar se um determinado valor está contido em um dicionário. Utilizamos o operador "in", que retornará True se objeto pertencer ao conjunto, e False caso contrário:

```
print('5GJ4:C' in dic) # Imprime True
```

Adicionalmente ao operador "in", podemos usar a versão complementar "not in” para verificar se a chave não está contida no dicionário.

Podemos obter o número de itens que compõem um dicionário mais uma vez através da função "len" assim como nas outras estruturas de dados compostos em Python:

```
dic = {'5GJA:A':61, '5GJ4:B':177, '5GJ4:C':61, '5GJK:D':177, '5GP1:B':268}
len(dic) # Imprime 5
```

Falemos agora dos métodos disponíveis para manipulação de objetos do tipo dicionário.

É comum na manipulação de dicionários a necessidade de unir um conjunto de elementos a um outro conjunto, ou seja, a construção da união dos conjuntos. Para isso, usamos o método “update”:

```
dic = {'5GJA:A':61, '5GJ4:B':177, '5GJ4:C':61, '5GJK:D':177, '5GP1:B':268}
dic2 = {'1A6M:A':252, '2MM1:A':251}
dic.update(dic2)
# dic = {'5GJA:A': 61, '5GJ4:B': 177, '5GJ4:C': 61, '5GJK:D': 177, '5GP1:B':
268, '1A6M:A': 252, '2MM1:A': 251}
```

O Python oferece as funções “min" e “max” através das quais é possível encontrar, respectivamente, a menor e a maior chave de um dicionário. Funciona para cadeias de caracteres, obviamente, e faz uma ordenação lexicográfica:

```
# dic = {'5GJA:A': 61, '5GJ4:B': 177, '5GJ4:C': 61, '5GJK:D': 177, '5GP1:B':
268, '1A6M:A': 252, '2MM1:A': 251}
max(dic) # Retorna “1A6M:A”
min(dic) # Retorna “5GP1:B”
```

O Python oferece o método “pop" para retornar e remover itens de dicionários:

```
# dic = {'5GJA:A': 61, '5GJ4:B': 177, '5GJ4:C': 61, '5GJK:D': 177, '5GP1:B':
268, '1A6M:A': 252, '2MM1:A': 251}
dic.pop('1A6M:A') # Retorna 252
# dic = {'5GJA:A': 61, '5GJ4:B': 177, '5GJ4:C': 61, '5GJK:D': 177, '5GP1:B':
268, '2MM1:A': 251}
```

Oferece ainda o método “popitem" para retornar e remover itens aleatórios de dicionários:

```
# dic = {'5GJA:A': 61, '5GJ4:B': 177, '5GJ4:C': 61, '5GJK:D': 177, '5GP1:B':
268, '1A6M:A': 252, '2MM1:A': 251}
dic.popitem() # Retorna, por exemplo, ('5GP1:B', 268)
# dic = {'5GJA:A': 61, '5GJ4:B': 177, '5GJ4:C': 61, '5GJK:D': 177, ‘1A6M:A':
252, '2MM1:A': 251}
```

Há situações em que a chave dos elementos que se deseja remover é desconhecida e desejamos remover o item a partir do seu valor e para tal, em listas, usamos o método “remove(item)”. Este método não funciona para dicionários.

Em alguns casos, pode ser necessário copiar um dicionários criando uma segunda versão dele. Temos que ter cuidado com a cópia de dicionários em Python pois o comando de atribuição de dicionários:

```
dic2 = dic
```

apenas copia a referência do dicionário "dic" para "dic2", ou seja, o nome "dic2" apontará para o mesmo endereço de memória para o qual aponta “dic”. Isso significa que quaisquer alterações feitas na estrutura / valores de “dic" se refletirá em "dic2" pois eles se tratam no mesmo objeto. Se desejamos realmente criar um outro objeto que seja uma cópia real devemos usar o método “copy”:

```
dic2 = dic.copy()
```

Nesse caso, outro bloco de memória (com outro endereço) será reservado para o novo dicionário "dic2" e todos os pares chave-valor serão copiados para essas novas posições de memória de forma que alterações em “dic" não refletirão em “dic2" e vice-versa.

Às vezes se faz necessária a conversão entre estruturas diferentes, como exemplo, converter um dicionário em uma lista. Há 3 métodos para extrair uma lista de um dicionário:

```
# dic = {'5GJA:A': 61, '5GJ4:B': 177, '5GJ4:C': 61, '5GJK:D': 177, '2MM1:A':
```

“keys" retorna a lista de chaves

```
# dict_keys(['5GJA:A', '5GJ4:B', '5GJ4:C', '5GJK:D', '2MM1:A'])
```

“values” retorna a lista de valores

```
# dict_values([61, 177, 61, 177, 251])
```

“items” retorna a lista de tuplas (pares chave-valor)

```
# dict_items([('5GJA:A', 61), ('5GJ4:B', 177), ('5GJ4:C', 61), ('5GJK:D',
177), ('2MM1:A', 251)])
```

É importante destacar que os os tipos dos objetos retornados não são listas propriamente ditas. “keys" retorna um objeto do tipo “dict_keys”, “values" retorna um objeto do tipo “dict_values” e “items” um objeto do tipo “dict_items”. Para utilizar esses objetos como listas podendo, por exemplo, fazer a indexação direta, é interessante convertê-los para listas usando coerção de tipo (usando o *cast list*):

```
# dic = {'5GJA:A': 61, '5GJ4:B': 177, '5GJ4:C': 61, '5GJK:D': 177, '5GP1:B':
268, '1A6M:A': 252, '2MM1:A': 251}
list(dic.keys())[0] # Retorna '5GJA:A'
```

Essas são algumas formas de extrair listas de dicionários. Mas será que em algum momento poderíamos querer extrair um dicionário de uma lista? Ou converter listas em dicionários?

Existe várias formas para que listas sejam convertidas em dicionários, no entanto, não podemos esquecer que um dicionário é um elemento que pares contém chave-valor. Não podemos converter uma única lista em um dicionário sem chaves, até seria possível o oposto, porém, não há muita razão para este tipo de abordagem. É mais comum convertermos duas listas que contém a mesma quantidade de elementos em um dicionário. Assim, se tivermos em uma lista um conjunto de elementos que representem as chaves e noutra um conjunto de elementos que representem os valores, nós podemos converter essas listas em dicionário. O método “zip" retorna uma lista contendo tuplas, onde o primeiro valor é o da primeira lista, e o segundo valor da tupla, corresponde a segunda lista. Com a função “zip" e a coerção de tipos, conseguimos obter um dicionário através de duas listas. No exemplo a seguir, criamos 2 listas, convertemos as listas em uma lista de tuplas e, finalmente, convertemos as listas em um dicionário:

```
aa3letras = ['ALA', 'CYS', 'ASP']
aa1letra = ['A', 'C', 'G']
list(zip(aa3letras, aa1letra))
# Retorna [('ALA', 'A'), ('CYS', 'C'), ('ASP', ‘G')]
dict(zip(aa3letras, aa1letra))
# Retorna {'ALA': 'A', 'CYS': 'C', 'ASP': 'G'}
```

Com relação à ordenação de dicionários, não faz sentido ordenar um dicionário visto que os elementos são armazenadas em posições não contíguas de memória. Podemos querer imprimir a lista de chaves e / ou valores de um dicionário em ordem alfabética ou de tamanho de chaves ou valores. Fazemos isso da seguinte forma:

```
# dic = {'5GJA:A': 61, '5GJ4:B': 177, '5GJ4:C': 61, '5GJK:D': 177, '2MM1:A':
251}
lista = list(dic.keys())
lista.sort()
print(lista) # Imprime ['2MM1:A', '5GJ4:B', '5GJ4:C', '5GJA:A', ‘5GJK:D']
```

Nomeando de variáveis

Há algumas regras importantes para nomear variáveis em Python. Elas devem iniciar com uma letra ou *underscore* e conter ao longo no nome letras, *underscores* e dígitos.

Escopo de variáveis

Até o momento, todas as variáveis que utilizamos são ditas **globais**, ou seja, são visíveis e acessíveis ao longo de todo o programa / script. Quando estudarmos funções e procedimentos, veremos que algumas variáveis podem ter o escopo **local**.

O escopo de uma variável indica sua visibilidade, ou seja, onde no código a variável é acessível.

Variáveis criadas dentro de uma função e que existem apenas dentro da função onde foi declarada. Elas são inicializadas a cada nova chamada à função e não é possível acessar seu valor fora da função onde ela foi declarada. Para que possamos interagir com variáveis locais, passamos parâmetros e retornamos valores nas funções.

Uma variável global é declarada (criada) fora das funções e pode ser acessada por todas as funções presentes no módulo onde é definida. Pode ser acessada por outros módulos, caso eles importem o módulo onde a variável foi definida. Uma aplicação útil de variáveis globais é o armazenamento de valores constantes no programa, acessíveis a todas as funções.

Retornaremos a esse assunto no final do módulo.

OPERADORES

Quando falamos em operadores em Python, nos referimos àqueles que já fazem parte da linguagem e vem implementados por padrão, não sendo necessário incluir nenhum módulo para sua utilização. Listamos alguns dos tipos de operadores mais utilizados em Python, bem como de outras linguagens.

Operadores aritméticos	
Adição	+
Subtração	-
Multiplicação	*
Divisão	/
Resto da divisão inteira	%

Operadores aritméticos	
Quociente inteiro da divisão	//
Exponenciação	**

Comparadores numéricos	
Igualdade	==
Desigualdade	!=
Menor que	<
Maior que	>
Menor ou igual que	<=
Maior ou igual que	>=
Dois operando se referem ao mesmo objeto	is
Pertence	in

Lógica booleana	
E	and
OU	or
NÃO	not

Diversos	
Atribuição	=
Concatenação de cadeias de caracteres	+ e *

Há alguns operadores que podem ser combinados com o "=" como por exemplo em "a += 2" que tem o mesmo efeito de "a = a + 2". Essa compressão da notação pode ser feita com "+", "-", "*" e "/".

Mostramos a seguir alguns exemplos de códigos com exemplos de uso desses operadores. Iniciando pelos operadores aritméticos

```
a = 4
b = 8

c = a + b # c=???
b = c - 2 # b=???
a = 23 * 9 # a=???
c = 100 / 9 # c=???
b = (a * b) # b=???
c = b % 2 # c=???
```

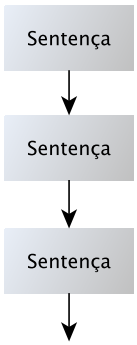
Desafio

Preencha os valores das variáveis (???) nos comentários do código anterior.

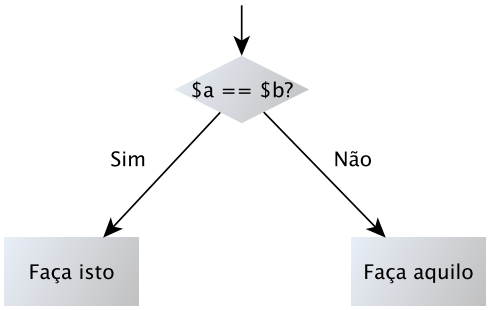
Os resultados serão respectivamente 12, 10, 207, 11.1111111111111, 2007, 0

ESTRUTURAS CONDICIONAIS

Até o momento, todos os exemplos que trabalhamos consistem em códigos estruturados e sequenciais. Se formos representá-los como fluxogramas, seria algo como ilustrado na seguinte figura:



Contudo, em programas reais, o tempo inteiro precisamos tomar decisões com base em algum tipo de condição.



If

Para implementarmos esse tipo de decisão toda linguagem de programação apresenta um comando chamado “if“ (se). Em Python, há algumas variações dos comandos condicionais que apresentaremos em exemplos a seguir.

```
if a == b:
        print('Variáveis são iguais')
```

If-else

No exemplo anterior, o comando só será impresso se a condição “a == b“ resultar em verdadeiro. Contudo, há uma limitação desse comando com relação ao tratamento dos casos em que a condição for falsa. Nesses casos, há uma variação do comando “if” que é o “if-else” (se-senão).

```
if a == b:
        print('Variáveis são iguais')
else:
        print('Variáveis não são iguais')
```

If-elif

Note que no caso do comando “if-else” o fluxo de execução passará obrigatoriamente pelo interior de um bloco ou do outro. Há casos em que várias condições devem ser testadas sequencialmente desde que as condições previamente testadas forem falsas. Nesse caso, Python possui a construção “if-elif“ (se-senão se). Veja o seguinte exemplo:

```
if a == b:
        print('Variáveis são iguais')
elif a > b:
        print('Variável a é maior que b')
elif a < b:
        print('Variável a é menor que b')
```

If-elif-else

Nesse exemplo anterior, em particular, uma e somente uma das três opções acima será verdadeira. Então, podemos usar esse exemplo para ilustrar ainda uma outra variação do comando condicional em Python que seria “if-elif-else“ (se-senão se-senão). Não há limite para o número de condições “elif“ que podem ser sequenciadas. Sempre deve haver um “if” inicial e o “else” final é opcional.

```
if a == b:
        print('Variáveis são iguais')
elif a > b:
        print('Variável a é maior que b')
else:
        print('Variável a é menor que b')
```

Veja o código a seguir e compare as duas opções apresentadas:

```
# Opção 1
if a == b:
        print('a e b são iguais')
if a != b:
        print('a e b são diferentes')

# Opção 2
if a == b:
        print('a e b são iguais')
elif a != b:
        print('a e b são diferentes')
```

Desafio

O que você pensa da eficiência em relação ao tempo computacional dessas duas versões da estrutura condicional apresentadas acima? Elas tem a mesma eficiência? Por que?

Switch

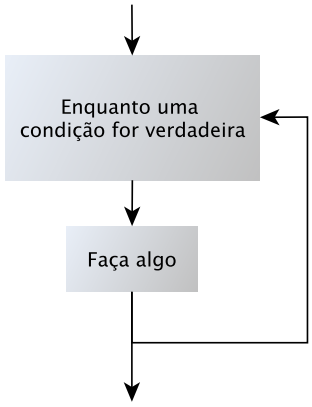
Grande parte das linguagens de programação possuem uma outra construção de comando condicional que é o seletor (do inglês *switch*). Ela consiste em usar uma variável cujo valor selecionará entre diversas opções de blocos de comandos. Python é uma linguagem projetada com o conceito de minimalismo e não apresenta a construção switch sem perda de expressividade. Você pode implementar o switch usando ifs, elifs e else.

ESTRUTURAS DE REPETIÇÃO

Você já se perguntou por que usamos programas de computadores?

Em geral, usamos programas de computadores para automatizar tarefas que sabemos fazer para ganharmos em **escala**. Um computador é capaz de executar em tempo curto bilhões de comandos não é mesmo? Então, implementamos algoritmos que formalizam as tarefas que precisamos e sabemos em detalhes como executar para que elas possam ser executadas em larga escala.

Para que essas tarefas possam ser executadas um certo número de vezes, precisamos de estruturas das linguagens de programação que nos permitam realizar repetições como por exemplo a do esquema abaixo. Chamamos essa estruturas de repetição e laços (do inglês *loops*).



Há algumas possibilidades de cenários nos quais podemos desejar expressar repetições:

- **Definidos**: sabemos previamente quantas vezes o laço irá executar.
- **Indefinidos**: não sabemos previamente quantas execuções serão realizadas e necessitamos testar a cada **iteração** (cada execução do laço) para decidirmos se o mesmo continua ou interrompe. Essa verificação pode ocorrer em dois momentos:
  - **Antes do laço**: nesse caso, pode acontecer de o bloco não ser executado nem uma vez (se a condição for falsa na primeira execução)
  - **No fim do laço**: garante que o laço será executado pelo menos uma vez, já que o teste da condição só é realizado ao término da primeira iteração.

Há também laços que nunca terminam de executar. Eles são conhecidos como **laços infinitos**. Podem ocorrem por um erro de lógica que não garante que uma condição de parada seja atingida. Em contrapartida, há casos em que eles são desejáveis no caso em que desejamos que o usuário encerre o programa manualmente ou que ele execute indefinidamente. Exemplo: um programa que fica rodando na rede monitorando algum comportamento suspeito do usuário.

Assim como com as estruturas condicionais, há diversas estruturas em Python para expressar repetições.

For

Começando por repetições definidas introduziremos a estrutura “for” (para):

```
nucleotideos = ['A', 'C', 'T', 'G']
for n in nucleotideos:
        print(n)
```

O código acima percorre toda a lista de nucleotídeos contida a lista “nucletideos" e atribui cada valor à variável “i” que é impressa no interior do laço. Embora hajam apenas 4 nucleotídeos, o mesmo trecho de código pode ser utilizado para listas de tamanho enorme.

Podemos iterar também sobre *strings*:

```
for n in 'ACCAGGAGGCGATG':
        print(n)
```

Em Python, com a instrução break, podemos parar o laço antes que ele tenha passado por todos os itens:

```
for n in 'ACCAGGAGGCGATG':
        print(n)
   if n == 'X':
        break
```

Nesse caso, quando encontrarmos um caracter “X”, pararemos a impressão da sequência. Observe agora os dois trechos de código a seguir e pense na diferença do resultado delas:

```
for n in 'ACCAGGAGGCGATG':
        print(n)
        if n == 'X':
                break
```

```
for n in 'ACCAGGAGGCGATG':
        if n == 'X':
                break
        print(n)
```

No trecho à esquerda, o “X” chegará a ser impresso enquanto no da direita, não.

Há ainda o comendo “continue”. Com essa instrução, podemos parar a iteração atual do laço e continuar com a próxima interação:

```
for n in 'ACCAGGAGGCGATG':
        if n == 'X':
                continue # Não imprimirá em caso de um nucleotídeo X
                         aparecer
```

Para percorrer um bloco de código um número pré-definido de vezes, podemos usar a função “range”. Ela retorna uma seqüência de números, iniciando em 0 (por padrão) e incrementa de 1 em 1 (por padrão) e termina com o número especificado.

```
for i in range(256):
        print(i) # Imprime de 0 a 255
```

Observe que o “range(10)" não são os valores de 0 a 10, mas os valores de 0 a 9. 0 é o valor padrão inicial, no entanto é possível especificar o valor inicial adicionando um parâmetro: "range(1,10)", que significa valores de 1 a 10 (mas não incluindo 10):

```
for i in range(100, 256):
        print(i) # Imprime de 100 a 255
```

Por padrão, a função range() incrementa o índice de 1 em 1 unidade. Contudo, é possível passar o valor do incremento como terceiro argumento:

```
for i in range(100,256,2):
        print(i) # Imprime de 100, 102, 104, 106, 108, …, 254
```

Em Python, existe ainda uma construção não usual que nos permite usar um "else" seguido de um laço “for”. O comando “else” em um laço "for" especifica um bloco de código a ser executado quando o laço for concluído, ou seja, quando a condição do “for” se torna falsa:

```
for n in 'ACCAGGAGGCGATG':
        print(n)
else:
        print('Fim do laço')
```

While

Os laços "while" (enquanto) são usados nos casos indefinidos, ou seja, naqueles em que não se sabe quantas vezes esse laço será executado e a condição de continuidade deve ser verificada a cada nova interação. Com o laço while, podemos executar um conjunto de instruções, enquanto a condição for verdadeira.

```
num = 289
divisor = num - 1

while num % divisor != 0 and divisor > 1:
        divisor -= 1

if divisor > 1:
        print(num, 'não é primo: divisível por', divisor)
else:
        print(num, ' é primo')
```

O laço "while" itera sobre o divisor e a condição de continuidade é que ambas as seguintes condições sejam verdadeiras:

- não ter encontrado um divisor: “num % divisor != 0" (resto da divisão inteira é diferente de zero)
- divisor é maior que 1

O laço vai decrementando o divisor e para de testar em duas situações

- encontrou um divisor do número
- testou todos os possíveis divisores e nenhum foi divisor exato

Laços aninhados

Um laço aninhado é um laço dentro de um laço. O laço interno será executado uma vez para cada iteração do laço externo.

```
seq = 'ACCAGGAGGCGATG'
for n in seq:
        for m in seq:
                print(n,m)
```

Trata-se de uma construção possível em todas as linguagens de programação e muito utilizada. Um exemplo de sua utilização é o do exemplo acima no qual desejamos “comparar” ou “tratar” todos os itens contra todos.

Laços infinitos

São laços que nunca terminam de executar. Eles podem ocorrer:

- por um erro de lógica que não garante que uma condição de parada seja atingida
- propositalmente nos casos em que desejamos que o usuário encerre o programa manualmente ou que ele execute indefinidamente. Exemplo: um programa que fica rodando na rede monitorando algum comportamento suspeito de usuários

Outros tipos de laços não existentes em Python

Há diversos outros tipos de construções que implementam laços em outras linguagens e não estão disponíveis em Python lembrando sua característica minimalista:

- Foreach (para cada) e versão tradicional do For (com as cláusulas de inicialização, condição e incremento)
- Do-While (Faça enquanto)
- Until (Até que)
- Do-Until (Faça até que)

ENTRADA E SAÍDA

Entrada e saída são operações de comunicação de um programa com o mundo externo. Há dois tipos básicos de entrada e saída que um programa pode realizar:

- por meio do teclado (pode ou não ser via linha de comando) e da tela
- por meio da escrita e leitura de arquivos gravados no disco rígido do computador

Em Python, a entrada e saída via linha de comando e tela também se dão através de arquivos que estão associados a dispositivos tais como discos rígidos, impressoras, teclados.

Usamos três arquivos padrão para tal. Toda vez que fazemos um print, fazemos um “sys.stdout”. Sempre que lemos um dado através do comando “readline” estamos lendo de um arquivo chamado “sys.stdin”. Mensagens de erro (não veremos nesse curso) são enviadas para o arquivo “sys.stderr".

Os arquivos “sys.stdin", “sys.stdout” e “sys.stderr" normalmente estão associados ao teclado e ao *display* do terminal sendo usado. O módulo "sys" do Python permite realizar algumas interações com o sistema operacional da máquina.

Pegando argumentos pela linha de comando

Uma das formas que podemos fazer a entrada de dados para o programa do mundo exterior é coletando argumentos ou parâmetros que podem ser passados ao lado do nome do programa no momento de sua execução (chamada):

```
~raquelcm$ python3 programa.py entrada.txt saida.txt
```

Neste exemplo estamos passando dois argumentos para o programa, um arquivo de entrada e um arquivo de saída, ambos arquivos de texto. Para coletar esses parâmetros vamos utilizar a lista “argv”, do módulo “sys,” assim:

```
import sys
for param in sys.argv:
        print(param)
```

Lendo do teclado em tempo de execução

Outra forma em que podemos fazer a entrada de dados para o programa do mundo exterior é coletando argumentos em tempo de execução do programa através de interação com o usuário via função “input”:

```
x = input('Digite o valor de x: ')
print(x)
```

Esse método é menos usado devido a Bioinformática normalmente lidar com dados em larga escala. Normalmente, usamos mais leituras de arquivos e fazemos pouca ou nenhuma interação com o usuário o que é mais indicado.

Abertura de arquivos

Em Bioinformática, trabalhamos com grande frequência com arquivos de dados biológicos tanto para leitura quanto para escrita de resultados. Em Python, você usará a função “open(nome, modo, buffering)” para abrir um arquivo tanto para leitura quanto para escrita sendo que

- **nome**: é o nome do arquivo a abrir
- **modo**: é o modo de abertura:
  - **r**: leitura
  - **w**: escrita
  - **b**: binário (há também a opção t para texto que é a padrão)
  - **a**: escrita a partir do final (anexar ou append)
  - **+**: leitura e escrita
- **buffering** (opcional): indica se memória (buffers) é usada para acelerar as operações e não será abordado no curso.

Uma vez que o arquivo tenha sido aberto, procedemos então à leitura dos dados ou a gravação. Certifique-se de passar um nome de arquivo existente ou obterá uma mensagem de erro.

Leitura

Uma vez que o arquivo tenha sido aberto, procedemos então à leitura dos dados ou a gravação. Comecemos pela leitura. Veja um exemplo de como o tratador de arquivo seria processado após a abertura do arquivo:

```
import sys
nomeArqEntrada = sys.argv[1] #Primeiro argumento passado via linha de comando
linhas = open(nomeArqEntrada, "rt")
for l in linhas:
        print(l)
```

Note que, por padrão, a função “print" de Python sempre imprime com quebra de linhas. Há duas variações que você pode usar desse código:

Tirar as quebras de linha de cada linha lida do arquivo:

```
for l in linhas:
        l = l.rstrip()
        print(l)
```

Imprimir sem a quebra de linha:

```
for l in linhas:
        print(l, end='')
```

Fechamento de arquivos

Após terminar de ler um gravar em um arquivo, ele sempre deve ser fechado através da função “close”. No caso do exemplo acima, o arquivo poderia ser fechado imediatamente ter o seu conteúdo copiado para a variável lista “linhas”.

```
import sys
nomeArqEntrada = sys.argv[1] #Primeiro argumento passado via linha de comando
linhas = open(nomeArqEntrada, 'rt')
for l in linhas:
        print(l)
linhas.close()
```

Pode-se utilizar a função “die" juntamente com o fechamento de arquivos.

Escrita

A escrita de arquivos é feita pela utilização do método "write":

```
import sys
nomeArqSaida = sys.argv[1] # Primeiro argumento passado via linha de comando
arq = open(nomeArqSaida, 'w')
arq.write('ACCGACCGATGCA')
arq.close()
```

O arquivo deve ter sido aberto para escrita (w ou a) Como se pode ver, escrever em arquivos é tão simples quanto imprimir na tela.

Remoção de arquivos e diretórios

Para remover um arquivo, precisamos usamos a função “remove" do módulo “os”:

```
import os
os.remove('arquivo.txt')
```

Se o arquivo não existir, você obterá um errode execução. Para evitar tal erro, melhor testar a existência do arquivo:

```
import os
if os.path.exists(‘arquivo.txt'):
        os.remove('arquivo.txt')
else:
        print('Arquivo inexistente')
```

Para remover um diretório, use a função “rmdir” passando como parâmetro o nome e o caminho do diretório

```
import os
os.rmdir('diretorio')
```

que você deseja remover. Por segurança, só é possível remover diretórios vazios.

Há opções mais avançadas para leitura e escrita em arquivos que não veremos nesse curso por limitações de tempo e por serem mais específicas. Acredito que a maioria dos casos de necessidade de tratamento de arquivos é coberta pelos métodos básicos abordados aqui. Alguns exemplos de operações mais avançadas são o tratamento de arquivos binários, a leitura e escrita no mesmo arquivo, leitura e escrita com *buffer* e escrita no meio do arquivo. Se algum desses métodos for necessário ao seu trabalho, recomendamos estudar cada caso em particular.

MODULARIZAÇÃO DE CÓDIGO

A programação modular é um conjunto de técnicas de projeto e organização de código que visa separar as diferentes funcionalidades de um sistema em módulos independentes e intercambiáveis. Normalmente pensamos nos módulos para que contenham todo o código necessário para implementação de uma funcionalidade.

Esse conceito é normalmente relacionado aos conceitos de programação estruturada e programação orientada por objetos que definimos a seguir:

- **Programação estruturada**: estruturação do fluxo de controle do programa tendo em vista a codificação (nível um pouco mais baixo)
- **Programação orientada por objetos**: uso de objetos de dados, um tipo de estrutura de dados

Programação modular é a decomposição em alto nível de um software em partes. Ela se diferencia da programação estruturada por essa quebra em módulos que normalmente são organizados internamente de forma estruturada. A programação orientada por objetos é modular mas nem toda programação modular é orientada por objetos.

Em Python, podemos modularizar código usado dois diferentes mecanismos que permitem organizar o sistema em **bibliotecas** e **funcionalidades**. Embora existam diferentes definições para esse termo, daremos nossa definição a seguir:

Biblioteca é uma coleção de implementações em alguma linguagem de programação e que pode ser utilizada por múltiplos programas que não necessariamente tenham relação uns com os outros possibilitando o que chamamos de **reuso**.

Funcionalidade é um conceito mais restrito é define uma tarefa indivisível do sistema e que normalmente possui uma **interface** bem definida composto por argumentos de **entrada** e valor(es) de **retorno** ou **saída**.

Vantagens

Podemos citar inúmeras vantagens de se modularizar código:

- Menos código a ser escrito
- Código escrito será mais simples e compreensível
- O escopo das variáveis é limitado e pode ser facilmente controlado
- O código fica organizado em arquivos separados de acordo com sua função
- Maior facilidade de projeto e divisão de trabalho entre equipes de desenvolvimento
- Uma funcionalidade em desenvolvimento pode ser reusada, eliminando a necessidade de se escrever o código múltiplas vezes
- Os erros são mais facilmente identificados, pois são localizados dentro de subrotinas.
- O software é mais fácil de manter e estender.

Como implementar modularização em Python

Começaremos pela modularização que não envolve a orientação por objetos, que será bordada no fim desse módulo. Assim, podemos implementar uma biblioteca através dos seguintes mecanismos:

Módulo que é um arquivo de código fonte, normalmente contendo funcionalidade de semântica relacionada. Exemplos: módulo de cálculos de medidas de média estatística, módulo de tratamento de sequências, módulo de sobreposição de estruturas, etc.

Subrotinas que também são denominadas **procedimentos** ou **funções** e que consistem em blocos de código que podem ser ativados (desviando o fluxo de execução do programa para si) quando necessário. Exemplos: subrotina para cálculo de média aritmética de um conjunto de valores, subrotina para alinhamento de um par de sequências, subrotina para obtenção de uma matriz de translação.

Uma subrotina é um termo genérico que pode se diferenciar segundo sua estrutura e utilidade:

- **Procedimento**: normalmente realiza uma tarefa mas não retorna um valor
- **Função**: usualmente retorna um valor ou um conjunto de valores. É análoga a uma função matemática na qual um valor é recebido na entrada e outro, gerado na saída.

Subrotinas

Veja a seguir um exemplo de definição e respectiva chamada a subrotina em Python:

Essa subrotina recebeu o nome de “minhaFuncao" e recebe 3 argumentos na chamada imprimindo-os. Ela não retorna nenhum valor como resultado.

```
# Definição da função
def minhaFuncao(a, b, c):
        print(a, b, c)

# Chamada da função
minhaFuncao(1, 2, 3)
```

Subrotinas também podem retornar um valor como no exemplo a seguir:

```
def quadrado(num):
        return num**2

print(quadrado(8)) # Imprimirá "64"
```

A função anterior recebe um número como argumento e retorna o seu quadrado, que é então impresso.

Escopo

Conforme introduzido anteriormente, o escopo de uma variável indica sua visibilidade – ou seja, a partir de onde, no código, a variável é acessível. Temos dois escopos para variáveis em Python:

- local
- global

As variáveis com escopo local são aquelas criadas dentro de blocos como funções, por exemplo. Existem apenas dentro do bloco onde foram declaradas, não sendo possível acessar seu valor fora do mesmo. São declaradas e inicializadas a cada nova chamada à função.

As variáveis com escopo global são declaradas (criadas) fora dos blocos, no programa principal, e podem ser acessadas por todas as funções presentes no módulo onde são definidas. Também podem ser acessadas por outros módulos, caso eles importem o módulo onde foram definidas. Uma aplicação útil de variáveis globais é o armazenamento de valores constantes no programa, acessíveis a todas as funções. É recomendado reduzir ao máximo o uso de variáveis globais em programas modulares pois dificultam o entendimento do código e vão contra o conceito de encapsulamento das rotinas, podendo ser alteradas por qualquer função, o que dificulta muitas vezes o rastreamento de quem as altera.

Passagem de parâmetro por valor e por referência

Agora que você já foi introduzido aos principais conceitos de modularização, subrotinas e módulos, entraremos em detalhes sobre as duas formas existentes de passagem de parâmetros para subrotinas. As linguagens de programação normalmente permitem que os argumentos ou parâmetros sejam passados de duas formas diferentes para as funções:

- **Passagem por valor**: forma mais comum e consiste em passar para a função uma cópia do valor de uma variável
  - O valor da variável será conservado enquanto durar seu escopo
  - É recomendável por evitar “efeitos colaterais"
    - Exemplo: família de funções trigonométricas, como seno, cosseno, etc.

```
def quadrado(num):
        num = num**2
        return num

num = 2
print(num) # Imprime 2
print(quadrado(num)) # Imprime 4
print(num) # Imprime 2
```

Esse é um exemplo de passagem de parâmetro por valor. Teste o código e diga o que será impresso ao final.

Note que será impresso o valor “2”. Por que isso acontece? Por que o valor de “num" que se torna “4" no interior da função não se conserva após o término da execução da mesma?

Isso acontece pois a variável “num" é copiada no momento da ativação da função ou seja, é criado um outro espaço de memória para essa variável e a mesma recebe o valor “4" e posteriormente, seu escopo termina com o término da execução da função na qual reside.

- **Passagem por referência**: consiste em passar para a função o endereço da variável
  - O valor da variável permanecerá mesmo após a desativação da função
  - É recomendável em alguns casos em que os “efeitos colaterais” sejam desejados como quando se deseja retornar na função mais de um valor
    - Exemplo: Retornar um número complexo (parte real e parte imaginária). Normalmente uma função não consegue retornar dois valores a menos que o façamos por meio de um arranjo de duas posições.

O exemplo seguinte ilustra a passagem por referência de uma lista “arr” que terá cada elemento multiplicado por 2 no interior da função “dobraArranjo”. O resultado da impressão será “2 4 6 8 10” indicando que a lista original foi modificada.

```
def dobraArranjo(arr):
        for i in range(0, len(arr)):
                arr[i] = arr[i]*2

arr = [1, 2, 4, 4, 5]
print(arr) # Imprime [1, 2, 3, 4, 5]
dobraArranjo(arr)
print(arr) # Imprime [2, 4, 6, 8, 10]
```

Sumário

- Em Python, todos os objetos são passados por valor
  - Mas tudo em Python são objetos!
- Isso implica que:
- Objetos imutáveis são passados por valor
- Objetos mutáveis (na prática) são passados por referência pois são referências! Assim, as referências são passadas por valor mas através delas, alteramos os dados dos objetos mutáveis como na passagem por referência.

Módulos

Para criar um módulo em Python, basta criar um arquivo com extensão “.py”. Esse módulo será posteriormente incluído através de “import <nome módulo>" em outros scripts “.py” que você desenvolver. Após incluir o módulo, você pode chamar funções que estejam implementadas no arquivo que foi incluído. Veja o exemplo a seguir:

```
# Arquivo sequencia.py
def conteudoGC(sequencia):
        s = list(sequencia)
        if len(s) == 0:
                print('Sequência vazia.')
                return -1
        cC = cG = 0
        for n in s:
                if n == 'C':
                        cC += 1
```

Esse é o conteúdo do módulo “sequencia.py”. Você pode incluir esse módulo em qualquer outro programa que fizer. Para usar a função em outro programa, você deve usar o comando "import" como no exemplo a seguir:

```
import sequencia
cGC = sequencia.conteudoGC('ACGTAGGGATGGCGTAGGAAAATGCGGGATGGCTGAGGCT')
print('%0.4f' % cGC)
```

Não há uma regra bem definida para guiar essa divisão de um programa em módulos e rotinas. Uma recomendação é segmentar o máximo que for possível. Sempre que você identificar uma porção do seu algoritmo que possa ser implementada dentro de uma rotina, você deve fazê-lo.

EXPRESSÕES REGULARES

Expressões regulares são mecanismos da linguagem de programação que lhe permitem de forma concisa e flexível identificar no texto caracteres particulares ou cadeias de caracteres de interesse ou ainda padrões mais complexos.

Esse termo expressão regular deriva do trabalho de Stephen Kleene que é um matemático que as desenvolveu com o nome de álgebra de conjuntos regulares e foi base para os primeiros algoritmos de busca e ferramentas de processamento de texto em Unix.

Com expressões regulares podemos

- **Casamento:** verificar se um texto apresenta casamento com um determinado padrão
- **Substituição:** substituir no texto caso ele apresente casamento com um padrão
- **Extração:** extrair partes de um texto caso ele apresente casamento com um determinado tipo de padrão.

Em Python, há um módulo nativo chamado "re" que é usado para trabalhar com expressões regulares e você precisará importá-lo sempre que desejar trabalhar com expressões regulares em seu programa.

```
import re
```

Vamos abordar a seguir, alguns dos métodos que estão disponíveis para expressões regulares.

Casamento

As principais funções para casamento ou busca de padrões é apresentada na seguinte tabela:

Função	Descrição	Retorno
findall	Retorna uma lista de casamentos encontrados	list
search	Retorna o objeto “casado”, se houver um casamento	**re.Match**
split	Retorna uma lista na qual a string foi cortada em cada casamento	list
sub	Substitui um ou muitos casamentos em uma string	string
finditer	Retorna um iterador sobre objetos do tipo casamento	**re.callable_iterator**

Você pode verificar, por exemplo, se uma sequência começa com “>”, ou seja, se é uma linha de cabeçalho em um arquivo fasta (por exemplo) através da função “search”:

Também poderia ser utilizada a função "match" que casa no início da *string*. A busca pelo caracter “>” no início da string é definida pelo metacaracter “^” que usamos no código acima. Assim como esse, existem inúmeros metacarteres que podemos usar para tornar nossa busca extremamente flexível e poderosa em termos dos padrões que podemos buscar. Um resumo dos principais metacaracteres é apresentada na tabela abaixo:

Caractere	Descrição
\A	Retorna uma correspondência se os caracteres estiverem no início da string
\b	Retorna uma correspondência se os caracteres estiverem início ou no fim da *string*
\B	Retorna uma correspondência se os caracteres estiverem presentes mas NÃO no início (ou no fim) da *string*
\d	Retorna uma correspondência onde a *string* contém dígitos (números de 0 a 9)
\D	Retorna uma correspondência onde a *string* NÃO contém dígitos
\s	Retorna uma correspondência onde a *string* contém um caractere de espaço em branco
\S	Retorna uma correspondência onde a *string* NÃO contém um caractere de espaço em branco
\w	Retorna uma correspondência onde a string contém qualquer caractere formador de palavras ou alfanumérivo (caracteres de a a Z, dígitos de 0 a 9 e o caractere de underscore _)
\W	Retorna uma correspondência onde a string contém qualquer caractere que não seja alfanumérico
\Z	Retorna uma correspondência se os caracteres especificados estiverem no final da string

Há ainda metacaracteres para representar conjuntos como de dígitos e / ou caracteres:

Caractere	Descrição
[ACTG]	Retorna uma corresponder onde um dos caracteres especificados (A, C, T ou G) está presente
[a-n]	Retorna uma correspondência para qualquer caractere minúsculo, alfabeticamente entre a e n
[^ACTG]	Retorna uma correspondência para qualquer caractere EXCETO A, C T e G
[0123]	Retorna uma correspondência onde qualquer um dos dígitos especificados (0, 1, 2 ou 3) estão presentes
[0-9]	Retorna uma correspondência para qualquer dígito entre 0 e 9
[0-5][0-9]	Retorna uma correspondência para quaisquer números de dois dígitos entre 00 e 59
[a-zA-Z]	Retorna uma correspondência para qualquer caractere em ordem alfabética entre a e z, minúscula OU maiúscula
[+]	Em conjuntos, +, *,., |, (), $, {} não tem significado especial, então [+] significa: retorna uma correspondência para qualquer caractere + na *string*

Vejamos, então, alguns exemplos de uso desses metacaracteres em conjunto com os principais métodos de expressões regulares para realizar algumas tarefas de interesse comum em bioinformática.

Encontrar todas as ocorrências do padrão “AA” na linha através do método “findall":

```
import re

linha = 'VLSAAEWQLVLHVWAAVEADVAGHGAAILIRLFKSHAATLEKFDRFKHLK'
x = re.findall('AA', linha)
print(x)
# Imprimirá
# ['AA', 'AA', 'AA', ‘AA’]
# OU
# [] se nenhum casamento for encontrado
```

Quebrar a sequência em pedaços pela ocorrência do padrão “AA" através do método “split”:

```
import re

linha = 'VLSAAEWQLVLHVWAAVEADVAGHGAAILIRLFKSHAATLEKFDRFKHLK'
x = re.split('AA', linha)
print(x)
# Imprimirá ['VLS', 'EWQLVLHVW', 'VEADVAGHG', 'ILIRLFKSH', 'TLEKFDRFKHLK']
```

Você pode inclusive controlar o número de "split"s executados se usar um parâmetro numérico a mais indicando o número máximo de casamentos que deseja realizar:

```
import re

linha = 'VLSAAEWQLVLHVWAAVEADVAGHGAAILIRLFKSHAATLEKFDRFKHLK'
x = re.split('AA', linha, 2)
print(x)
# Imprimirá ['VLS', 'EWQLVLHVW', 'VEADVAGHGAAILIRLFKSHAATLEKFDRFKHLK']
```

Substituição

Você pode utilizar o método “sub” para substituir as ocorrências um padrão. Por exemplo, substituir todas as ocorrências de “AA" por “—“:

```
import re

linha = 'VLSAAEWQLVLHVWAAVEADVAGHGAAILIRLFKSHAATLEKFDRFKHLK'
x = re.sub('AA', '--', linha)
print(x)
print(linha)
# Imprimirá
# VLS--EWQLVLHVW--VEADVAGHG--ILIRLFKSH--TLEKFDRFKHLK
# VLSAAEWQLVLHVWAAVEADVAGHGAAILIRLFKSHAATLEKFDRFKHLK
```

De forma análoga ao que se faz usando a função "split", você também pode controlar quantas substituições fazer no máximo passando um parâmetro inteiro a mais:

```
import re

linha = 'VLSAAEWQLVLHVWAAVEADVAGHGAAILIRLFKSHAATLEKFDRFKHLK'
x = re.sub('AA', '--', linha, 2)
print(x)
print(linha)
# Imprimirá
# VLS--EWQLVLHVW--VEADVAGHGAAILIRLFKSHAATLEKFDRFKHLK
# VLSAAEWQLVLHVWAAVEADVAGHGAAILIRLFKSHAATLEKFDRFKHLK
```

Objeto do tipo Match

Você deve ter notado que o método “search” não retorna um valor de tipos primitivos mas um objeto do tipo “Match". Um objeto do tipo "Match" contém informações sobre a pesquisa e o resultado. Se não houver correspondência, o valor "NONE" será retornado, em vez do objeto. O objeto tem propriedades e métodos que são usados para recuperar informações sobre a busca e sobre o resultado como:

- "span": retorna uma tupla contendo as posições inicial ("start") e final ("end") do casamento. Cuidado: a posição final vem sempre somada de uma unidade!
- *string*: a string passada para a função (a *string* em que se faz a busca)
- “group": retorna a parte da *string* onde houve o casamento
- a *string* casada

Veja então um exemplo de utilização de uma variável “x" desse tipo:

```
import re

linha = 'VLSAAEWQLVLHVWAAVEADVAGHGAAILIRLFKSHAATLEKFDRFKHLK'
x = re.search('AA', linha)
print('O primeiro AA encontrado foi na posição', x.start())
# Imprimirá "O primeiro AA encontrado foi na posição 3”
# ou "O primeiro AA encontrado foi na posição None” se não for encontrado
```

E veja ainda um exemplo em que recuperamos todas as ocorrências de um padrão em um objeto do tipo iterados de “match”s:

```
import re

linha = 'VLSAAEWQLVLHVWAAVEADVAGHGAAILIRLFKSHAATLEKFDRFKHLK'
for i in re.finditer('AA', linha):
    print(linha[i.start():i.end()], i.start(), i.end())
# Imprimirá
# AA 3 5
# AA 14 16
# AA 27 29
```

Busca gulosa x Busca preguiçosa

Os qualificadores "*", "+" e "{}" que vimos previamente são chamados gulosos. Eles buscam a mais distante correspondência possível. Acontece que , às vezes esse comportamento não é o desejado. Adicionar "?" após o qualificador faz com que a busca seja preguiçosa, ou seja, a correspondência mais próxima será retornada. Veja como isso pode ser útil para implementar um *web scraper*:

```
import re

linha = '<h1>Título</h1>'
x = re.search('<.*>', linha)
print(x.group())
# Imprimirá "<h1>Título</h1>”
x = re.search('<.*?>', linha)
print(x.group())
# Imprimirá "<h1>”
```

CLASSES E OBJETOS

Não faz parte do programa do nosso curso, abordar a teoria de programação orientada por objetos. Esse conteúdo poderia ser abordado em um curso inteiro sobre programação modular. Com relação à orientação por objetos, em particular, os conceitos de encapsulamento, herança, polimorfismo, entre outros, não serão trabalhados no nosso curso por questão de tempo e escopo.

Contudo, Python é uma linguagem que permite a aplicação de alguns dos conceitos de orientação por objetos por possuir mecanismos de criação de classes e objetos e por possibilitar a herança entre classes. Não entraremos nesse curso nos detalhes das vantagens de aplicação de orientação por objetos (que são vários) mas trataremos apenas da sintaxe em Python para criação de classes e objetos. Quase tudo em Python é um objeto, com suas propriedades e métodos. Uma classe é um construtor de objetos ou um molde para criar objetos.

Vamos ao primeiro exemplo em que criamos uma classe chamada "MinhaClasse", com uma propriedade chamada “x”:

```
class MinhaClasse:
        x = 5
```

Após definir uma classe, podemos usar a mesma para criar objetos desse tipo:

```
obj1 = MinhaClasse()
print(obj1.x)
```

O método construtor __init__()

Para entender o significado das classes, precisamos entender a função construtora denominada em Python com o nome padrão “__init__()”. Todas as classes têm uma função "__init__()", que é sempre executada quando a classe está sendo instanciada (criação de um objeto desse tipo). Use a função "__init__()" para atribuir valores a propriedades do objeto ou outras operações necessárias ao objeto que está sendo criado:

```
class Proteina:
        def __init__(self, nome, tamanho):
                self.nome = nome
                self.tamanho = tamanho

p1 = Proteina('Hemoglobina Humana', 252)
print(p1.nome)
```

Classes também podem conter métodos. Métodos são rotinas implementadas dentro das classes e que podem ser ativadas para cada objeto desse tipo. Vamos criar um método que imprime uma mensagem na classe “Proteina":

```
class Proteina:
        def __init__(self, nome, tamanho):
                self.nome = nome
                self.tamanho = tamanho

        def minhaFuncao(self):
                print('Proteína: ' + self.nome)

p1 = Proteina('Hemoglobina Humana', 252)
p1.minhaFuncao()
```

O parâmetro self

O parâmetro "self" é uma referência à instância atual da classe (objeto atual) e é usado para acessar variáveis que pertencem à classe. Ele não precisa ser nomeado "self", você pode chamá-lo como quiser, mas tem que ser o primeiro parâmetro de qualquer função na classe:

```
class Proteina:
        def __init__(eu, nome, tamanho):
                eu.nome = nome
                eu.tamanho = tamanho

        def minhaFuncao(euMesma):
                print('Proteína: ' + euMesma.nome)

p1 = Proteina('Hemoglobina Humana', 252)
p1.minhaFuncao()
```

Você pode modificar propriedades em objetos diretamente através de atribuições assim:

```
p1.tamanho = 255
```

Esse é um ponto importante. Em Python, não existe controle de acesso / visibilidade em atributos de classes o que é uma desvantagem pois não permite a implementação da diferenciação entre variáveis públicas, protegidas e privadas. Isso faz com que alguns conceitos da teoria de orientação por objetos não possam ser assegurados. O que existe em Python é uma convenção de que nomes de variáveis que desejamos que sejam protegidas sejam nomeadas com o nome antecedido por um "_", como por exemplo "_x", e antecedido por "__" para as privadas.

Você pode deletar propriedades em objetos usando a palavra chave del:

```
del p1.tamanho
```

Por fim, é muito importante que você libere a memória que não será mais necessária quando um objeto não for mais ser utilizado, deletando os objetos inteiros usando a palavra chave "del" seguida pelo nome do objeto:

```
del p1
```

O foco principal desse módulo do nosso curso é apresentar a programação básica em Python. Quando você considerar que absorveu os conceitos básicos, que está conseguindo expressar a lógica de programação em Python, prossiga seus estudos para conceitos mais intermediários como a orientação por objetos. Será importante se você deseja desenvolver sistemas de médio e maior porte para organização do seu código, facilitar o reusabilidade, a manutenção e extensão dos seus programas. Além disso, se você pretende usar bibliotecas de terceiros como a BioPython, por exemplo, será muito importante você se aprofundar mais em orientação por objeto em Python.

TRATAMENTO DE EXCEÇÕES

Tratamento de exceções é o mecanismo responsável pelo tratamento de condições inesperadas ou errôneas que precisam alterar o fluxo normal da execução de programas. Em geral, na ocorrência de uma exceção, o estado do programa é gravado em um local pré-definido e a sua execução é direcionada para uma rotina de tratamento. Dependendo da situação, a rotina de tratamento pode prosseguir com a execução a partir do ponto que originou a exceção, utilizando a informação gravada para restaurar o estado. Lançar uma exceção é um modo útil de assinalar que a rotina não deve continuar a execução e um procedimento de tratamento do erro deve ser realizado.

Exemplos de exceções que devem ser tratadas são argumentos de entrada inválidos. Se o seu programa não pode prosseguir sem os dados corretos de entrada, ele não deve prosseguir a menos que se consiga obter esses dados. Por exemplo, a ocorrência de um denominador igual a zero em uma divisão é algo que precisa ser tratado. Uma possibilidade é pedir ao usuário que digite novamente o denominador. Outra é simplesmente imprimir o erro de forma amigável e interromper definitivamente a execução do programa. Outro erro importante é um recurso do qual o programa depende e que pode não estar disponível, como um arquivo não encontrado.

Em Python, usamos, principalmente, três comandos em conjunto que criam blocos responsáveis por lançar e tratar exceções:

- o bloco "try" permite testar um bloco de código quanto a erros
- o bloco "except" permite que você lide com o erro
- O bloco "finally" permite executar o código, independentemente do resultado dos blocos "try" e “except"

Quando ocorre uma exceção, o Python normalmente pára e gera uma mensagem de erro. Essas exceções podem ser lançadas em um bloco de código passível de erro delimitado pela instrução "try":

```
try:
        print(x)
except:
        print('Uma exceção ocorreu.')
```

Como o bloco "try" gera um erro, o bloco “except" será executado. Sem o bloco "try", o programa falha e é interrompido gerando um erro.

Você pode definir quantos blocos de exceção desejar, por exemplo se você deseja executar um bloco de código especial para um tipo especial de erro:

```
try:
        print(x)
except NameError:
        print('Variável indefinida.')
except:
        print('Uma exceção ocorreu')
```

Você pode ainda usar a palavra-chave "else" para definir um bloco de código a ser executado se nenhum erro for gerado:

```
try:
        print(x)
except:
        print('Ocorreu alguma exceção.')
else:
        print('Nenhuma exceção ocorreu')
```

Por fim, veja um exemplo final que inclui todas essas possibilidades de blocos para tratamento de exceções:

```
try:
        f = open('arquivo.txt')
        f.write('teste')
except:
        print('Algo deu errado com o arquivo')
finally:
        f.close()
```

Assim como alertamos na seção anterior, sobre orientação por objetos, o conceito de tratamento ode exceções é muito importante para a construção de sistemas robustos. Por ser um tópico também intermediário, recomendamos ao estudante que desejar desenvolver sistemas de porte maior estudarem esse conceito posteriormente no momento oportuno.

Referências

[Site Python] https://www.python.org/

[Site Python Documentation] https://docs.python.org/3/

[Celes et al., 2004] Celes, Waldemar, Renato Cerqueira e José Lucas Rangel. Introdução a Estruturas de Dados: com técnicas de programação em C. Elsevier, 2004.

[Mathieu e Gillings, 2008] Mathieu, Fourment e Gillings, Michael R.. "A comparison of common programming languages used in bioinformatics." BMC bioinformatics 9.1 (2008): 82.

[Ziviani, 2004] Ziviani, Nivio. Projeto de algoritmos: com implementações em Pascal e C. Vol. 2. Thomson, 2004.