Introdução à Programação em Python aplicada à Bioinformática

Paulo J. Martel
2015
mestradocbm1415@gmail.com

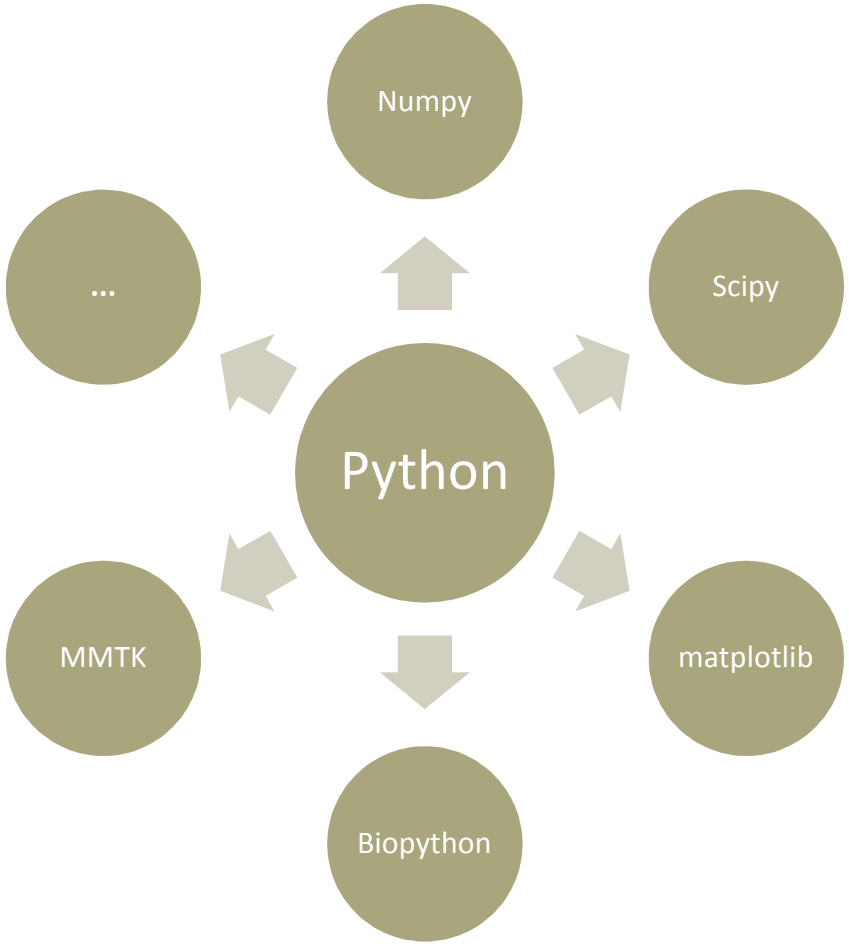
O que é o Python ?

- Linguagem de programação criada em 1989 por Guido van Rossum
- Disponível para uma variedade de sistemas informáticos, incluindo Windows, Mac, Linux e outros sistemas operativos.
- Plataforma de utilizadores crescente
- Grande variedade de aplicações científicas
- Gratuita
- Fácil de aprender, no entanto poderosa
- O nome “Python” é uma homenagem do autor ao grupo de comédia britânica “Monty Python”

Porquê a linguagem Python?

- Interpretada (visualização imediata dos resultados)
- Fácil de aprender
- Gratuita
- Disponível para múltiplas plataformas informáticas
- Inclui *múltiplos paradigmas* de programação
- Linguagem de alto nível
- Elevada legibilidade do código
- Elevada funcionalidade *intrínseca*
- Elevada *portablidade* do código
- Extensível através de um enorme número de módulos, muitos dos quais de aplicação científica

Extensibilidade



Grande variedade de módulos disponível

Legibilidade

- Código em Python:

```
print "Hello World!"
```

- Código em Java:

```
public class Hello
{
	public static void main(String[] args) {
		System.out.printf("Hello World!");
	}
}
```

Porquê programar?

- Automatização de tarefas repetitivas
- Realização de pequenas tarefas para as quais não existe software disponível
- Criação de novas aplicações ou métodos
- Extender a funcionalidade de um sistema de software
- Acesso especializado a recursos online

Um programa é uma receita

Protocol for Restriction Digestion of Lambda DNA

Materials:
5.0 µL Lambda DNA
2.5 µL 10x Buffer
16.5µL H20
1.0 µL EcoRI

Procedure:

Incubate the reactions at 37 C for one hour
Add 2.5 uL loading dye and incubate another 15 min
Load 20 uL of the digestion mixture onto a minigel

Um programa é uma receita

```
# Program to calculate the average of a set of numbers
sum=0
count=0
count=input(“How many numbers?”)
For a in range(count) do:
      sum = sum + input(“Insert a number...”)

average = sum / count
print “The average of this set of %d number is %d” %
(count,sum)
```

Conceitos fundamentais

- O interpretador de Python
  - aritmética em Python
- Variáveis
  - Declaração de variáveis
- Tipos de estruturas de dados
  - Números, strings, listas, dicionários
- Estruturas de controle
  - **if**, **if else, for**, **while**
- Interacção com o utilizador
  - **input**, **raw_input**
- Operações sobre ficheiros
  - **open**, **close,** ...

O interpretador de Python

```
>>> print “hello world”
hello world
>>> 2+2
4
>>> 52*345
17940
>>> 2.0 / 3.0
0.6666666666666666
>>> a=1
>>> b=2
>>> a+b
3
>>> for a in range(5):
        print a,
0 1 2 3 4
>>>
```

Tipos de dados em Python

- Numéricos:
  - Integer (1,2,3,-1,-2,0,7,234,...)
  - Float (1.23,0.3566,3.53e+63)
  - Complex (2+3j,2j,5+27j)
  - Boolean (True, False)

- Estruturados:
  - Strings ('ATGCCCAATTG')
  - Listas ([1 ,5 , 0.2, 'A', 'xxx'])
  - Tuplas ((1, 2, 3))
  - Conjuntos (set(['A','T','G','C']))
  - Dicionários ({ 'A' : 'Ala', 'V' : 'Val', 'I' : 'Ile', 'L' : 'Leu'})

Ordenados (sequenciais)

Não Ordenados

Operações numéricas

Símbolo	Descrição
+	Soma
-	Subtracção
*	Multiplicação
/	Divisão
**	Exponenciação
%	Resto da divisão (módulo)

Exemplos

```
>>> 12323*3242
39951166
>>> 10**20
10000000000L
>>> 2147483647+1
2147483648L
```

```
>>> 10/4
```

```
>>> 10.0/4
```

Notar a diferença entre divisão inteira e real!

```
>>> 2.345**300
1.1037078090771378e+111
Comparar:
>>> 15.0**400
>>> 15**400
```

Variáveis

- Podem conter letras de A-z (maiúsculas e minúsculas)
- Podem conter números
- Podem conter o símbolo "_"
- Não podem começar por um número
- Não podem conter símbolos com significado especial em Python - *, %, -, +, &, ...
- Distinção entre maiúsculas e minúsculas:

```
>>> EcoRI = 1
1
>>> print ecoRI
Traceback (most recent call last):
  File "<pyshell#334>", line 1, in <module>
    print ecori
NameError: name 'ecori' is not defined
```

Variáveis

```
>>> 1aa = 12
SyntaxError: invalid syntax

>>> Ala+ = 10
SyntaxError: invalid syntax
>>> new_seq = 'ATTTGGC'
>>> Ala = 10
>>> Ala+new_seq
Traceback (most recent call last):
  File "<pyshell#345>", line 1, in <module>
    new_seq+Ala
TypeError: cannot concatenate 'str' and 'int' objects
>>> str(Ala)+new_seq
'10ATTTGGC'
```

Strings

- Representação de uma cadeia de caracteres, imutável

```
>>> seq1='ATGGGCA'
>>> seq2='AATTAAAT'
>>> seq1 + seq2
'ATGGGCAAATTAAAT'
>>> poly_alanine='A'*100
>>> len(poly_alanine)
100
>>> poly_alanine
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
>>> seq1[0]
'A'
>>> seq1[3:6]
'GGC'
```

Strings (operations)

Operação	Descrição
x in s	**True** se x pertence a s
x not in s	**True** se x não pertencer a s
s + t	Concatenação
s*n	Produz *n* repetições de *s*
s[i]	i-ésimo elemento de s
s[i:j]	Slice: elementos de i a j
s[i:j:k]	Slice: elementos de i a j com intervalo k
len(s)	Comprimento de s
min(s)	Menor elemento de s*
max(s)	Maior elemento de s*

* Implica a existência de uma relação de ordem entre os elementos de s

Strings (operations)

```
>>> seq1
'ATGGGCA'
>>> 'A' in seq1
True
>>> 'X' in seq1
False
>>> len(seq1)
7
>>> min(seq1)
'A'
>>> max(seq1)
'T'
>>> seq1*3
'ATGGGCAATGGGCAATGGGCA'
```

Strings (methods)

Método	Descrição
str.capitalize()	Primeira letra para maiúscula
str.count(**s**)	Conta número de ocorrências de **s**
str.index(**s**,[*start*, [*end*]])	Retorna a posição de **s**
str.replace(**s**,**r**)	Subsitui **s** por **r**
str.split(**sep**)	Separa numa lista usando separador **sep**
str.lower()	Converte para minúsculas
str.strip()	Remove espaços circundantes
str.isalpha()	**True** se str é apenas alfabético
str.upper()	Convere para maísculas

Strings (methods)

```
>>> myseq = 'ATTGGCCAAACCG'
>>> myseq.count('A')
4
>>> myseq.count('CC')
2
>>> myseq.replace('A','U')
'UTTGGCCUUUCCG'
>>> myseq.capitalize()
'Attggccaaaccg'
>>> myseq.lower()
'attggccaaaccg'
>>> myseq.index('AAA')
7
>>> '193.136.227.168'.split('.')
['193', '136', '227', '168']
>>>
```

Listas

- As listas são um dos tipos de estrutura mais poderoso e versátil do Python. São um tipo *sequencial* como os strings, mas são mutáveis

Exemplos:

```
>>> my_list = [1,2,3,4]
>>> other_list = ['A','T','G','C']
>>> my_list+other_list
[1, 2, 3, 4, 'A', 'T', 'G', 'C']
>>> my_list[2]
3
>>> my_list[2]=100   # as listas são mutáveis!
>>> my_list
[1, 2, 100, 4]
>>> my_list[2]= other_list   # qual será o resultado ?
[1, 2, ['A', 'T', 'G', 'C'], 4]
```

Operações em listas

Operação	Descrição
l.append(x)	Adiciona o element x à lista l
l.count(x)	Conta o número de ocorrências de x em l
l.index(x)	Retorna a posição de x em l
l.insert(index,x)	
l.remove(x)	Remove o elemento x da lista l
l.reverse(x)	Inverte a ordem dos elementos da lista l
l.sort(x)	Ordena os elementos da lista l
l.pop(i)	Remove o elemento na posição i da lista l

Dicionários

- Os dicionários são um tipo de estrutura *não-ordenada* que permite associar pares de chaves (keys) e valores (values)

Formato: { *key1*:*value1, key2*:*value2, key3*:*value3, ...* }

Exemplo:

```
>>> aa_code = {'A':'Ala', 'G':'Gly', 'V':'Val', 'L':'Leu'}
>>> print "The code for amino %s is %s" % ('A', aa_code['A'])
The code for amino A is Ala
>>> aa_code['V']
'Val'
>>> aa_code['W'] = 'Trp'
>>> aa_code.keys()
['A', 'W', 'L', 'G', 'V']
>>> aa_code.values()
['Ala', ‘Trp', 'Leu', 'Gly', 'Val']
```

Dicionários (operações)

Método	Descrição
len(a)	Número de elementos em a
a[k]	Elemento a com chave k
a[k] = v	Associa o valor v com a chave k
del a[k]	Elimina a entrada com chave k
a.clear()	Limpa o dicionário a
k in a	**True** se k for uma das chaves de a
k not in a	**True** se k não for uma das chaves de a
a.items()	Gera uma lista de pares (key, value)
a.keys()	Lista de keys de a
a.values()	Lista de valore de a
a.get(k[,x])	Retorna a[k], ou x se k não estiver em a
a.pop(k[,x])	a[k] ou x se k não estiver em a, remove k

Input / Output

- **raw_input** – permite ler variáveis introduzidas pelo utilizador, para uma variável tipo **string**

```
>>> seq = raw_input("Introduza uma sequência: ")
Introduza uma sequência: ATTGGCCCGAA
>>> print seq
ATTGGCCCGAA
```

Problema:

```
>>> n = raw_input("Introduza um número: ")
Introduza um número: 5
>>> print "o quadrado do número introduzido é ", n*n
o quadrado do número introduzido é

Traceback (most recent call last):
  File "<pyshell#507>", line 1, in <module>
    print "o quadrado do número introduzido é ", n*n
TypeError: can't multiply sequence by non-int of type 'str'
```

Input / Output

- O uso da função **int()** resolve o problema:

```
>>> n = int(raw_input("Introduza um número: "))
Introduza um número: 5
>>> print "o quadrado do número introduzido é ", n*n
o quadrado do número introduzido é  25
```

- Uma solução mais prática é usar a função **input()**

```
>>> n = input("Introduza um número: ")
Introduza um número: 5
>>> print "o quadrado do número introduzido é ", n*n
o quadrado do número introduzido é  25
```

- **input** – lê um **string** submetido pelo utilizador, e usa o avaliador de expressões do Python para calcular o seu resultado (tal como quando escrevemos uma expressão na *consola* do Python)

NOTA – Esta solução é perigosa, pois permite passar um programa ao interpretador de Python, por exemplo executando software malicioso

Input / Output

- O comando **print** pode ser usado para produzir "output" *formatado* de forma específica

```
>>> x=55.23456
>>> print "O valor de x é %5.2f" % x
O valor de x é 55.23
>>> print "O valor de x é %08.3f" % x
O valor de x é 0055.235
```

Forma geral de **print** formatado:

```
print "... %n.mc ... %n.mc ... " % (x,y,z,...)
```

FORMATOS VARIÁVEIS

Formatos

Método	Descrição
%**n**d	*Inteiro* num campo de **n** caracteres
%**n**s	*String* num campo de **n** caracteres
%**n.m**f	*Float* , **n** caracteres, **m** dígitos de precisão
%**n.m**g	*Float* , **n** caracteres, **m** dígitos de precisão
%**n.m**e	*Float*
%**n**x	*Inteiro* hexadecimal , campo de **n** caracteres
%**n**o	*Inteiro* octal, campo de **n** caracteres
%**n**c	*Inteiro* num campo de **n** caracteres
%%	Produz um um único '%'

Exercícios

- Escrever um programa que traduz uma sequência de DNA em mRNA
- Escrever um programa que calcula o conteúdo GC de uma sequência de DNA

```
# Converting a DNA sequence to mRNA

dna_seq = “ATTGGGAAAAACCCGTCTTACGGG”
rna_seq = dna_seq.replace('T','U')
print "A sequência traduzida é: ", rna_seq
```

```
# Calculate the G+C percentage content of a DNA sequence

dna_seq = "ATTAGGGATTTAAATTGAATCFGGCGCCCAGGGGGCCCAAT"

g_count = dna_seq.count('G')
c_count = dna_seq.count('C')
length = len(dna_seq)

gc_count = g_count + c_count

gc_content = 100.0*gc_count / length

print "Sequencia de DNA: ",dna_seq
print 'Conteudo GC: %.2f' % gc_content
```

Expressões booleanas

Expressão	Descrição
a < b	**True** se a for menor que b
a <= b	**True** se a for menor ou igual a b
a > b	**True** se a for maior que b
a >= b	**True** se a for maior ou igual a b
a == b	**True** se a for igual a b
a != b ou a <> b	**True** se a for diferente de b
not a	**True** se a for **False**
a or b	**True** se a for **True** ou b for **True**
a and b	**True** se a for **True** e b for **True**

No caso de não se verificarem as condições indicadas, o valor da expressão será **False**

Expressões booleanas

```
>>> 1 > 2
False
>>> not 1 > 2
True
>>> a = 2
>>> a == 2
True
>>> (a>0) and (a<3)
True
>>> (a>0) or (a<-100) #Qual será o resultado ?
True
```

N.B. : não confundir estas duas situações – no primeiro caso temos uma atribuição de valor a uma variável, no segundo um **teste** lógico de igualdade

Estruturas de controle

- Comandos da linguagem Python que permitem controlar o **fluxo** do nosso programa
- **Condicionais:** o programa é executado se determinada condição for verdadeira (**if**, **else**)
- **Ciclos:** determinada região do programa é repetida um número de vezes pré-determinado pelo utilizador, ou até determinada condição se verificar (**for**, **while**)

Ciclos com **for**

- A estrutura de um ciclo **for** é a seguinte:

for VAR **in** ITERABLE:
Block

em que a variável VAR assume todos os valores possíveis de ITERABLE (que poder ser uma lista, string, etc...)

Exemplo:

```
>>>for i in [1,2,3,4]:
       print i,
1 2 3 4
>>>
```

Ciclos com **for**

- Para iterar um número específico de ciclos com **for** o comando range(min, max, step) é muito útil:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(3,12,2)
[3, 5, 7, 9, 11]
```

```
>>> for i in range(1,8):
        print i,
3 4 5 6 7
```

- O comando **xrange()** é uma versão mais eficiente de **range()**, recomendada para intervalos muito grandes (Ex: xrange(100000))

Exercício

- Escrever um programa que calcula a massa molecular de uma proteína a partir da sua sequência (http://pastebin.com/xqsX23Wi)

```
protseq = raw_input("Enter your protein sequence: ")
protweight = {"A":89,"V":117,"L":131,"I":131,"P":115,"F":165,
              "W":204,"M":149,"G":75,"S":105,"C":121,"T":119,
              "Y":181,"N":132,"Q":146,"D":133,"E":147,
              "K":146,"R":174,"H":155}
totalW = 0
for aa in protseq:
    totalW = totalW + protweight.get(aa.upper(),0)
totalW = totalW-(18*(len(protseq)-1))
print "The net weight is: “, totalW
```

Condições com **if** e **else**

- A estrutura de uma condição **if-else** é a seguinte

```
if EXPRESSION:
        Block1
else:
        Block2
```

Exemplo:

```
>>> a=2
>>> if a > 1:
        print "é maior que 1"
else:
        print "não é maior que 1"


é maior que 1
>>>
```

Condições com **if** e **elif**

- A estrutura de uma condição **if-else** é a seguinte

```
if EXPRESSION:
        Block1
elif EXPRESSION2:
        Block2
elif EXPRESSION3:
        Block3
else:
        Block4
```

Condições com **if** e **elif**

- Exemplo:

```
dna = raw_input("Enter your DNA sequence: ")
seqsize = len(dna)
if seqsize < 10:
    print("Your primer must have at least 10 nucleotides")
    if seqsize == 0:
        print("You must enter something!")
elif seqsize < 25:
    print("This size is OK")
else:
    print("Your primer is too long")
```

Exercício:

- Escrever um programa que verifica se uma sequência de DNA é válida, e indica a posição dos caracters inválidos

```
# testing if a DNA sequence is valid

dna_seq = raw_input("Input your DNA sequence: ").upper()

i = 1
Invalid_char = False
for bb in dna_seq:
    if bb not in 'ATGC' :
        print "Invalid character %s in sequence at position %d" %
(bb,i)
        Invalid_char = True
    i = i + 1

if Invalid_char:
    print "Your sequence contains invalid characters."
else:
    print "Your sequence is valid."
```

Ciclos com **while**

- O comando **while** executa um bloco de comandos até se verificar um determinada condição

```
while EXPRESSION:
        BLOCK
```

Exemplo:

```
>>> a=10
>>> while a < 40:
        a = a + 10
10
20
30
```

Exercício

- Escrever um programa que converte uma sequência de protéina de código de uma para 3 letras.
  http://pastebin.com/fkLm3dve

```
protseq = raw_input("Enter protein sequence: ").upper()

AAcodes =
{'A':'Ala','G':'Gly','V':'Val','L':'Leu','I':'Ile','M':'Met','F':'Phe',
'Y':'Tyr','W':'Trp',
'C':'Cys','H':'His','D':'Asp','E':'Glu','K':'Lys','R':'Arg','S':'Ser','
T':'Thr','N':'Asn','Q':'Gln',
'P':'Pro'}

i=0
while i < len(protseq)-1:
    print AAcodes[protseq[i]]+" -",
    i = i + 1
print AAcodes[protseq[i]]
```

Funções

- As funções são uma forma de tornar os programa *modulares*
- Uma função recebe zero, um ou mais valores, executa uma determinada acção e pode retornar um valor como resultado
- Forma geral da declaração de uma função:

```
def Nome(argumento1, argumento2,...):
        """Descrição opcional da função"""
        BLOCK
        return DATA
```

Exemplo:

```
>>> def mul(n):
        """Multiplica um número por 10"""
        return n*10

>>> mul(3)
30
>>> help(mul)
```

Exercício:

- Escrever uma função que calcula o factorial de um número

  (n! = n*(n-1)*(n-2)*(n-3)* ... *1)

  5! = 5*4*3*2*1

```
>>> def fact(n):
        """Calcula o factorial de n"""
        f = 1
        while n > 1:
                f = n * f
                n = n - 1
        return f

>>> fact(5)
120
>>> help(fact)
```

Exercício:

- Escrever uma função que calcula percentagem de GC numa sequência de dna

```
>>> def GC(seq):
        """Calcula a %GC de uma sequência de DNA"""
        g = seq.count('G')
        c = seq.count('C')
        return (g+c)/float(len(seq))

>>> print GC(‘ATTGACCATTGGCCA’)
```

Leitura/Escrita de Ficheiros

A leitura e escrita de ficheiros pode ser feita com a função **open**, cuja forma geral é

filehandle = open(*filename*,*mode*)

mode: 'r' (read), 'w' (write), 'a' (append), 'r+´ (read and write
filename: nome do ficheiro
filehandle: referência do ficheiro

Métodos suportados por *filehandle*:

filehandle.read(n)	lê n bytes do ficheiro
filehandle.readline()	lê uma linha do ficheiro
filehandle.readlines()	lê as linhas do ficheiro para uma lista
filehandle.write('*string*')	escreve o *string* no ficheiro
filehandle.seek(*n*)	vai para a posição *n* do ficheiro
filehandle.close()	fecha o ficheiro

Leitura/Escrita de Ficheiros

Exemplo:

```
>>> fh = open('file.txt','r')
>>> fh.readlines()
['This is the first line\n', 'This is the second line\n', 'This
is the third line\n', '123\n', '456\n', 'This is the last
line\n']
>>> fh.readlines()
[]
>>> fh.seek(0)      # volta ao início do ficheiro
>>> fh.readline()
'This is the first line\n'
>>> fh.readline()
'This is the second line\n'
>>> fh.readline()
'This is the third line\n'
```

Exercício

- Escrever um program que lê um ficheiro em formato FASTA

```
# Program that reads a fasta file
fh = open('seqA.fasta')
FirstLine = fh.readline()
name = FirstLine[1:-1]
sequence = ""
while True:
    line = fh.readline()
    if line == "":
        break
    sequence += line.replace('\n','')

print "The name is ", name
print "The sequence is ", sequence
```

Classes

- As categorias de dados até agora discutidas (*int, float, string, list, dict*) permitem representar muitos tipos de variáveis, listas e relações, mas não permitem a descrição de *objectos* mais complexos.
- Uma classe é uma especificação da estrutura de um determinado tipo de objectos
- Uma *instância* é um objecto que pertence a uma determinada class
- As classes podem conter variáveis, bem como *atributos* e *métodos.*

Classes

- Forma geral de declaração de uma classe:

```
class Nome([parent_class]):
        """Descrição opcional da classe"""
        variable1 = …
        variable2 = …
        def __init__(self, var1, var2, …):
                self.attribute1 = ….
                self.attribute2 = ….
                ……..
        def method1(self,var1,var2,….):
                ……..
        def method2(self,var1,var2,….):
                ……..
```

Classes

- Exemplo simples:

```
class Rectangle(object):
    """
This class defines rectangles.
    """

    def __init__(self, a, b):
        self.a, self.b = a, b

    def area(self):
        return self.a*self.b

    def perimeter(self):
        return 2*(self.a+self.b)
```

Classes

- Construir uma classe sequência com um método de tradução:

```
class Sequence:
   TranscriptionTable = {"A":"U","T":"A","C":"G","G":"C"}

def __init__(self, seqstring):
       self.seqstring = seqstring.upper()
def transcription(self):
       tt = ""
       for x in self.seqstring:
          if x in 'ATCG':
             tt += Sequence.TranscriptionTable[x]
       return tt
```

Extender o Python: módulos

- A funcionalidade do Python é extendida integrando módulos com o comando **import**, ou **from** ... **Import**

import *module* [**as** *name*]
from *module* **import** *function* [**as** *name*]

Exemplo:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(0.2)
0.19866933079506122
>>> math.asin(0.19866933079506122)
0.2
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']
>>> math.sqrt(2)
1.4142135623730951
```

A biblioteca Biopython

- Biopython é uma biblioteca de módulos de Python destinados ao desenvolvimento de aplicações bioinformáticas

Módulo	Descrição
Bio.Alphabet	Alfabetos para proteínas e DNA/RNA
Bio.Seq	Definição de sequências biológicas
Bio.Align	Manipulação de alinhamentos
Bio.SeqIO	Leitura e escrita de sequências
Bio.Blast	Interface para o programa BLAST
Bio.Data	Vários tipos de dados biológicos
Bio.Entrez	Interface para o portal ENTREZ
Bio.SeqUtils	Ferramentas para sequências
Bio.PDB	Leitura e análise de ficheiros PDB
Bio.Prosite	Interface com PROSITE
Bio.Restriction	Sites de restrição em DNA
Bio.SubstMat	Alignment matrices
Bio.pairwise2	Sequence alignment

Bio.Alphabet

```
>>> import Bio.Alphabet
>>> Bio.Alphabet.ThreeLetterProtein.letters
['Ala', 'Asx', 'Cys', 'Asp', 'Glu', 'Phe', 'Gly', 'His',
'Ile', 'Lys', 'Leu', 'Met', 'Asn', 'Pro', 'Gln', 'Arg',
'Ser', 'Thr', 'Sec', 'Val', 'Trp', 'Xaa', 'Tyr', 'Glx']
>>> from Bio.Alphabet import IUPAC
>>> IUPAC.IUPACProtein.letters
'ACDEFGHIKLMNPQRSTVWY'
>>> IUPAC.unambiguous_dna.letters
'GATC'
>>> IUPAC.ambiguous_dna.letters
'GATCRYWSMKHBVDN‘
>>> IUPAC.ExtendedIUPACProtein.letters
'ACDEFGHIKLMNPQRSTVWYBXZJUO'
```

Bio.Seq

- Este módulo permite a definição e manipulação de sequências de DNA e proteína

```
>>> from Bio.Seq import Seq
>>> import Bio.Alphabet
>>> seq = Seq('CCGGATTGGAC',Bio.Alphabet.IUPAC.unambiguous_dna)
>>> seq
Seq('CCGGATTGGAC', IUPACUnambiguousDNA())
>>> seq.transcribe()
Seq('CCGGAUUGGAC', IUPACUnambiguousRNA())
>>> seq.translate()
>>> Seq('PDW', IUPACProtein())
>>> len(seq)
11
>>> seq+seq
Seq('CCGGATTGGACCCGGATTGGAC', IUPACUnambiguousDNA())
```

Bio.Seq

- As sequências são imutáveis, mas podem ser convertidas em objectos mutáveis com o método **.tomutable**

```
>>> seqm = seq.tomutable()
>>> seqm
MutableSeq('CCGGATTGGAC', IUPACUnambiguousDNA())
>>> seqm.reverse()
>>> seqm
MutableSeq('CAGGTTAGGCC', IUPACUnambiguousDNA())
>>> seqm.complement()
>>> seqm
MutableSeq('GTCCAATCCGG', IUPACUnambiguousDNA())
>>> seqm.reverse_complement()
>>> seqm
MutableSeq('CCGGATTGGAC', IUPACUnambiguousDNA())
```

Bio.SeqUtils

- Exemplo: calcular a “melting temperature” de uma sequência de DNA e o seu “GC content”

```
>>> from Bio.SeqUtils import MeltingTemp
>>> MeltingTemp.Tm_staluc('tgcagtacgtatcgt')
42.21147274487345
>>> print '%.2f'%MeltingTemp.Tm_staluc('tgcagtacgtatcgt')
42.21
>>> from Bio.SeqUtils import GC
>>> GC('gacgatcggtattcgtag')
50.0
```

Bio.SeqIO

- Exemplo: ler um conjunto de ficheiros em formato FASTA

```
from Bio import SeqIO
fh = open("seqs.fasta")
for record in SeqIO.parse(fh, "fasta"):
    id = record.id
    seq = record.seq
    print("Name: %s, size: %s"%(id, len(seq)))
fh.close()
```

Bio.Data

- Código genético mitocondrial (vertebrados)

```
>>> from Bio.Data import CodonTable
>>> print CodonTable.generic_by_id[2]
Table 2 Vertebrate Mitochondrial, SGC1

  |  U      |  C      |  A      |  G      |
--+---------+---------+---------+---------+--
U | UUU F   | UCU S   | UAU Y   | UGU C   | U
U | UUC F   | UCC S   | UAC Y   | UGC C   | C
U | UUA L   | UCA S   | UAA Stop| UGA W   | A
U | UUG L   | UCG S   | UAG Stop| UGG W   | G
--+---------+---------+---------+---------+--
C | CUU L   | CCU P   | CAU H   | CGU R   | U
C | CUC L   | CCC P   | CAC H   | CGC R   | C
C | CUA L   | CCA P   | CAA Q   | CGA R   | A
C | CUG L   | CCG P   | CAG Q   | CGG R   | G
--+---------+---------+---------+---------+--
A | AUU I(s)| ACU T   | AAU N   | AGU S   | U
A | AUC I(s)| ACC T   | AAC N   | AGC S   | C
A | AUA M(s)| ACA T   | AAA K   | AGA Stop| A
A | AUG M(s)| ACG T   | AAG K   | AGG Stop| G
--+---------+---------+---------+---------+--
G | GUU V   | GCU A   | GAU D   | GGU G   | U
G | GUC V   | GCC A   | GAC D   | GGC G   | C
G | GUA V   | GCA A   | GAA E   | GGA G   | A
G | GUG V(s)| GCG A   | GAG E   | GGG G   | G
--+---------+---------+---------+---------+--
```

Bio.pairwise2

- Calcula o alinhamento entre duas sequências

```
>>> from Bio import pairwise2
>>> alignments = pairwise2.align.globalxx("ACCGT", "ACG")
>>> from Bio.pairwise2 import format_alignment
>>> for a in pairwise2.align.globalxx("ACCGT", "ACG"):
...     print(format_alignment(*a))
ACCGT
|||||
AC-G-
  Score=3
<BLANKLINE>
ACCGT
|||||
A-CG-
  Score=3
<BLANKLINE>
```

Bio.SubsMat

- Matrizes de alinhamento (PAM, Blosum, Gonnet, …) e funções auxiliares

```
>>> from Bio.SubstMat.MatrixList import pam250
>>> print pam250[('W':'W')]
17
>>> from Bio.pairwise2 import format_alignment
>>> for a in pairwise2.align.globaldx("KEVLA", "EVL",pam250):
        print format_alignment(*a)
KEVLA
|||||
-EVL-
  Score=14
```

Bio.Restriction

- Encontrar sites de restrição em sequências de DNA

```
>>> from Bio import Restriction
>>> Restriction.EcoRI
EcoRI
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet.IUPAC import IUPACAmbiguousDNA
>>> alfa = IUPACAmbiguousDNA()
>>> seqi = Seq('CGCGAATTCGCG',alfa)
>>> Restriction.EcoRI.search(seqi)
[5]
>>> Restriction.EcoRI.catalyse(seqi)
(Seq('CGCG', IUPACAmbiguousDNA()), Seq('AATTCGCG',
IUPACAmbiguousDNA()))
>>> enz1=Restriction.EcoRI
>>> enz2=Restriction.HindIII
>>> batch1 = Restriction.RestrictionBatch([enz1,enz2])
>>> batch1.search(seqi)
{EcoRI: [5], HindIII: []}
```

Bio.Entrez

- Pesquisar no pubmed

```
from Bio import Entrez
my_em = 'user@example.com'
db = "pubmed"
# Search de Entrez website using esearch from eUtils
# esearch returns a handle (called h_search)
h_search = Entrez.esearch(db=db, email=my_em,
                          term="python and bioinformatics")
# Parse the result with Entrez.read()
record = Entrez.read(h_search)
# Get the list of Ids returned by previous search
res_ids = record["IdList"]
# For each id in the list
for r_id in res_ids:
    # Get summary information for each id
    h_summ = Entrez.esummary(db=db, id=r_id, email=my_em)
    # Parse the result with Entrez.read()
    summ = Entrez.read(h_summ)
    print(summ[0]['Title'])
    print(summ[0]['DOI'])
    print('==============================================')
```

Bio.Entrez

- Pesquisar no gene bank

```
from Bio import Entrez
my_em = 'user@example.com'
db = "gene"
term = 'cobalamin synthase homo sapiens'
h_search = Entrez.esearch(db=db, email=my_em,term=term)
record = Entrez.read(h_search)
res_ids = record["IdList"]
for r_id in res_ids:
    h_summ = Entrez.esummary(db=db, id=r_id, email=my_em)
    summ = Entrez.read(h_summ)
    print(r_id)
    print(summ[0]['Description'])
    print(summ[0]['Summary'])
    print('==============================================')
```


NEXT ...
NEXT ...
NEXT ...
NEXT ...
AM I MISSING SOMETHING? THIS IS SUSPICIOUSLY EASY ...