Python praktikum

Alessandro Mammana

10.10.2014

Max Plank Institute for Molecular Genetics

Praktikum layout

- introduction and “finger exercises” (2-3 hours)
- let's go to eat something
- guided problem (1-2 hours)
- free time (until 18:00)

Python: pros and cons

- Pros:
  - easy to write and to read
  - interactive
  - general-purpose programming language
- Cons:
  - not as fast as C, C++ and Java (but faster than R)
  - not suitable for very large projects

Easy to write and to read

```
def sort(array):
    if (len(array) <= 1):
        return array

    less = []
    equal = []
    greater = []

    pivot = array[0]
    for x in array:
        if x < pivot:
            less.append(x)
        if x == pivot:
            equal.append(x)
        if x > pivot:
            greater.append(x)

    return sort(less)+equal+sort(greater)
```

Python is interactive

```
~> python3
```

```
Python 3.3.0 (default, Dec  5 2012, 11:05:54)
[GCC 4.7.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello Python!")
Hello Python!
>>>
```

Python is general-purpose

- You are not going to use python only for bioinformatics
- Applications:
  - very good for scripting
  - scientific computing (numpy and scipy)
  - developing web applications (django)
  - plugins of many applications can be written in python (e.g. inkscape)

Interactive computing, scripting, programming

1. Interactive computing

```
~> python3
```

```
Python 3.3.0 (default, Dec  5 2012, 11:05:54)
[GCC 4.7.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello Python!")
Hello Python!
>>> quit()
```

2. Scripting:

```
~> mkdir pyprakt
~> cd pyprakt
~/pyprakt> gedit code.py &
```

```
print(“hello world!”)
```

```
~/pyprakt> python3 code.py
hello world!
```

or:

```
#!/usr/bin/python3
print(“hello world!”)
```

```
~/pyprakt> ./code.py
hello world!
```

Interactive computing, scripting, programming

3. Programming

```
import sys #importing module
def imHappy(n):  #function definition
   for i in range(n): #indent code using spaces or a tab
      print("Hello Python!")

if __name__=="__main__": #make the python program runnable from terminal
    imHappy(int(sys.argv[1]))
```

```
~/pyprakt> python3 code.py 3
"Hello Python!"
"Hello Python!"
"Hello Python!"
~/pyprakt> python3
```

```
Python 3.3.0 (default, Dec  5 2012, 11:05:54)
[GCC 4.7.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import code
>>> code.imHappy(3)
"Hello Python!"
"Hello Python!"
"Hello Python!"
```

No coding, no learning

- we will play with python in interactive mode
- we will write functions in gedit (or any text editor)
- keep 2 windows open, one for python's terminal, one for the editor
- how to modify/add functions in scratchpad.py and reload them:

```
import sys
def imHappy(n):
   for i in range(n):
      print("Good morning Python!") #we modified the code!

if __name__=="__main__":
    imHappy(sys.argv[1])
```

```
>>> import imp
>>> imp.reload(code) #reload an already imported module, don't forget it!
>>> code.imHappy(3)
"Good morning Python!"
"Good morning Python!"
"Good morning Python!"
```

Python as a calculator

- Syntax as most other programming languages
- two main numeric types: int and float
- powers with the ** operator
- import the math module for more advanced functions

```
>>> 2 + 2 # int + int = int
4
>>> 50 - 5*6
20
>>> (50 - 5.0*6)/4 # float
5.0
>>> 8/5.0 #float
1.6000000000000001
>>> 8//5 #integer division
1
>>> 17 // 3.0 #explicit integer division
5.0
>>> 5 * 3 + 2  # result * divisor + remainder
17
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
>>> import math
>>> math.sqrt(40)
6.324555320336759
>>> math.cos(2*math.pi)
1.0
```

Assigning to a variable

- the equal sign (=) is used for variable assignment
- variables need to be defined before they are used
- multiple assignment

```
>>> width = 20
>>> height = 5*9
>>> width*height
900
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
>>> a, b, c = 1, 2, 3  #multiple assignment
>>> b
2
>>>
```

Warming up

compute the following number: $\cos\left(\pi \frac{12^{1.5}+4^2+3^3}{42}\right)$

```
>>> math.cos(math.pi*(12**(1.5) + 4*4 + 3**3)/42)
0.9990937147385679
```

Sequence types

Arrays in different flavours

- Strings: sequence of characters
- Lists: mutable sequence of arbitray objects
- Tuples: immutable sequence of arbitrary objects
- Ranges: implicit sequence of consecutive numbers

They all support random access (like arrays) and many other things

```
>>> msg = "Python : <3"
>>> msg[0]
'P'
>>> msg[1]
'y'
>>> len(msg)
11
```

Strings

To define a string

- Enclose it in single quotes 'text'
- Enclose it in double quotes “text”
- if the text contains also quotes, escape them with \
- strings are immutable (cannot be modified)

```
>>> 'coding like a boss'  #single quotes
'coding like a boss'
>>> 'doesn\'t' #use \ to escape the single quote
"doesn't"
>>> "doesn't" #no need to escape single quotes inside double quotes
"doesn't"
>>> '"Yes," he said.' # no need to escape double quotes inside single quotes
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> secret = "this is not going to work"
>>> secret[0] = 'W'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Lists

- General-purpose array
- Comma-separated items between square brackets
- Elements can have different types
- lists are mutable
- to add an element, use the append function
- lists can be nested to any depth

```
>>> a = ['bio', 'inf', 100, 1234]
>>> a
['bio', 'inf', 100, 1234]
>>> a[1] = 'madness'  #lists are mutable
>>> a
['bio', 'madness', 100, 1234]
>>> a.append("rock'n'roll")  #append function
>>> a
['bio', 'madness', 100, 1234, "rock'n'roll"]
>>> a.append(['another', 'list']) #nested list
>>> a
['bio', 'madness', 100, 1234, "rock'n'roll", ['another', 'list']]
>>> b = [2, 'nested', 'lists:', a]
>>> b
[2, 'nested', 'lists:', ['bio', 'madness', 100, 1234, "rock'n'roll", ['another', 'list']]]
>>> b[3][1] #nesting list is a way of creating matrices
'madness'
```

Tuples

Difference compared to lists:

- round brackets instead of square brackets
- immutable
- can be converted to lists using the list() function

```
>>> a = ('bio', 'inf', 100, 1234)
>>> a
('bio', 'inf', 100, 1234)
>>> a[1] = 'madness'  #lists are immutable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a = list(a)
>>> a[1] = 'madness'
>>> a
['bio', 'madness', 100, 1234]
```

The beauty of slicing with python

Python supports very powerful slicing (subsetting) operators

- to count elements from the end, use negative indices

```
>>> mystr = '12345'
>>> mystr[-1] #same as mystr[len(mystr)-1], but much nicer!
'5'
>>> mystr[-3]
'3'
>>> mylist = [1,2,3,4,5]
>>> mylist[-3]
3
```

- to subset use [start:end]
- start defaults to 0, end defaults to the sequence length
- can also specify a step [start:end:step]

```
>>> mytuple = ("don't", "slice", "me", "please")
>>> mytuple[1:3] #start is included, end excluded
('slice', 'me')
>>> mytuple[1:-1] #negative indices can be mixed with positives
('slice', 'me')
>>> mylist[2:] #same as mylist[2:len(mylist)]
[3, 4, 5]
>>> mylist[:2] #same as mylist[0:2]
[1, 2]
>>> mylist[:] #same as mylist[0:len(mylist)] <- good for copying sequences
[1, 2, 3, 4, 5
```

Slicing and combining

- with the + operator, strings, tuples and lists can be combined
- with the * operator, strings, tuples and lists can be repeated
- with lists, whole slices can be replaced

```
>>> "that's " + "very " + "intuitive" # + for combining sequences
"that's very intuitive"
>>> mystr = "sliced and combined like a toy"
>>> mystr[:6] + mystr[6:]
'sliced and combined like a toy'
>>> mystr[11:19] + mystr[6:11] + mystr[:6] + mystr[19:]
'combined and sliced like a toy'
>>> "that's " + "very "*3 + "intuitive" # * for repeating sequences
"that's very very very intuitive"
>>> [1]*1 + [2]*2 + [3]*3 + [1,2,3]*3
[1, 2, 2, 3, 3, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> surprise = [1,2,3,4]
>>> surprise[3:] = [1,4,9] #that's not the same as: surprise[3] = [1,4,9]
>>> surprise
[1, 2, 3, 1, 4, 9]
>>> surprise[2:-2] = ["who", "put", "me", "there?"]
>>>
>>> surprise
[1, 2, 'who', 'put', 'me', 'there?', 4, 9]
```

Finger exercises

- we store paths to fasta files in python strings. Every path ends in ".txt". Write a function that takes in such a path and changes its ending in ".fasta"
- Example:

```
>>> fixEnding("path/to/myfasta.txt")
path/to/myfasta.fasta
```

```
def fixEnding(fapath):
    return fapath[:-3] + "fasta"
```

- we store reads from a sequencing experiment in python strings. read quality often gets worse at the ends of the read. Write a function that takes as input a string and a number n and cuts away the first n and the last n bases/characters.
- Example:

```
>>> cutReadEnds("AAATACGTGAAACATAAA", 3)
"TACGTGAAACAT"
```

```
def cutReadEnds(read, n):
    return read[n:-n]
```

- write a function that creates a list with n zeros
- Example

```
>>> zeros(4)
[0,0,0,0]
```

```
def zeros(n):
    return [0]*n
```

Finger exercises

- Write a function that takes in a list and returns a tuple with two lists: the even elements and the odd elements of the original list
- Example:

```
>>> mylist = ['one', 1, 'two', 2, 'three', 3, 'four', 4]
>>> magicSplit(mylist)
(['one', 'two', 'three', 'four'], [1, 2, 3, 4])
```

```
def magicSplit(mylist):
        return (mylist[::2], mylist[1::2])
```

Booleans

- 'True' or 'False' keywords
- They work as you would expect
- boolean operators are more readable than in almost all other languages
- boolean operators convert any object to a boolean
- the keyword 'None' is normally to encode special values, it is converted to 'False'
- Empty sequences are also converted to 'False', anything else behaves like 'True', but it's not converted

```
>>> 1>4  # comparisons return booleans
False
>>> 1>4 or 5>4   # 'or', 'and' are much more readable than '||' and '&&' (Java, C, C++, R)
True
>>> False and 5>4
False
>>> False or 5>4
True
>>> None or False # None gets converted to False
False
>>> [] or False # empty sequence gets converted to False
False
>>> "" or False
False
>>> [3,2] or False # anything else behaves like True, but it's not converted
[3, 2]
>>> "anything else" or False
'anything else'
```

If statements and while loops

- Just be careful to the intendation
- 'else if' in python is 'elif'
- the 'while' loop is how you expect

```
>>> if True:
...     print("I love Python") #careful to the indentation!
I love Python
>>> if False:
...     print("Python is hard to learn")
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
More
>>> a, b = 0, 1 #Fibonacci series
>>> while b < 10: #while loop, again, it must be indented!
...     print(b)
...     a = b
...     b = a+b
...
1
2
4
8
```

for loop

- The 'for' statement loops over the elements of a sequence
- How to get the “traditional” for loop over a range of numbers?
  - the range() function creates a range of consecutive numbers (efficiently)

```
>>> for i in 'string, list, tuple, range...\n':
...     print(i, end="") #variable 'i' takes on all the characters in the sequence
string, list, tuple, range...
>>> nums = [1,2,3,4]
>>> for i in nums:
...     print(i**2)
1
4
9
16
>>> for i in range(3): #traditional foor loop
...     print("Hello Python")
Hello Python
Hello Python
Hello Python
>>> for i in range(len(nums)):
...     nums[i] = nums[i]**2
>>> nums
[1, 4, 9, 16]
>>> #in java:
>>> #for (int i = start; i < end; i+=step){
>>> #     print(i)
>>> #}
>>> #in python:
>>> start, end, step = 0, 6, 2
>>> for i in range(start, end, step):
...     print(i)
```

Finger exercises

- Write a function that takes in two numbers 'a' < 'b' and returns a list with all consecutive numbers from 'a' to 'b' ('b' escluded) squared
- Example:

```
>>> squareList(1, 4)
[1, 4, 9]
```

```
def squareList(a, b):
    res = []
    for i in range(a, b):
        res = res + [i*i]
    return res

#or... better and faster with append()
def squareList(a, b):
    res = []
    for i in range(a, b):
        res = res.append(i*i)
    return res

#or... making a large-enough list before the loop
def squareList(a, b):
    res = list(range(a,b))
    for i in range(len(res)):
        res[i] = i*i
    return res
```

Finger exercises

- Write a function that takes to integers 'nrow' and 'ncol' and returns a matrix of zeros with nrow rows and ncol columns. Implement the matrix as a list of lists, where the nested lists are the rows.
- Example:

```
>>> makeMat(2, 4)
[[0,0,0,0], [0,0,0,0]]
```

```
def makeMat(nrow, ncol):
    mat = []
    for i in range(nrow):
        row = []
        for j in range(ncol):
            row.append(0)
        mat.append(row)
    return mat
#we can do it in less lines!
def makeMat(nrow, ncol):
    mat = []
    for i in range(nrow):
        mat.append([0]*ncol)
    return mat
#but this is going to be wrong... why?
def makeMat(nrow, ncol):
    return [[0]*ncol]*nrow
```

```
>>> mat = makeMat(2,4)          #the multiplication operator copies by reference,
>>> mat                         #when it needs to copy a non-primitive type
[[0, 0, 0, 0], [0, 0, 0, 0]]    #int is primitive
>>> mat[0][0] = 1               #list is not
>>> mat                         #Copying an immutable type (e.g. a string) is safe
[[1, 0, 0, 0], [1, 0, 0, 0]]
```

Finger exercises

- The factorial (in German Fakultaet) function is defined like this:

  0! = 1

  n! = 1*2*3*...*n

- write a function that takes an integer n and returns the factorial
- Example:

```
>>> fact(0)
1
>>> fact(4)
24
>>> fact(35)
265252859812191058636308480000000
```

```
#Solution 1:
def fact(n):
    res = 1
    for i in range(1,n+1):
        res *= i
    return res

#Solution 2:
def fact(n):
    res = 1
    while (n > 0):
        res *= n
        n -= 1 #in python there is no n++ or n--
    return res
```

Finger exercises

- The Euclid algorithm computes the greatest common divisor between two stictly positive integers 'a' and 'b'. It goes like this:
  - 1. if 'b' is equal to zero, the result is 'a', otherwise
  - 2. compute the remainder between 'a' and 'b'
  - 3. replace 'a' with 'b'
  - 4. replace 'b' with the remainder just computed, go back to step 1
- Example:

```
>>> gcd(3, 7)
1
>>> gcd(21, 28)
7
```

```
def gcd(a, b):
    while (b != 0):
        tmp = a % b
        a = b
        b = tmp
    return a
```

List comprehension

- very often we need to create a list with a for loop
- because Python requires to indent the code, we need at least 3 lines of code
- but Python also has a solution to this: list comprehension
- it reduces 3 lines to only 1, and it's readable!!!
- the statement can be extended to have nested for loops and an if statement

```
>>> mylist = []
>>> for i in sequence:
...   mylist.append(fun(i))
>>>
>>> mylist = [fun(i) for i in sequence]
>>>
>>> mylist = []
>>> for a in seq1:
...   for b in seq2:
...     for c in seq3:
...       if test(a, b, c):
...         mylist.append(fun(a, b, c))
>>>
>>>  mylist = [fun(a,b,c) for a in seq1 for b in seq2 for c in seq3 if test(a,b,c)]
```

Finger exercises

- rewrite the functions squareList and makeMat in one line

```
def squareList(a, b):
    return [i*i for i in range(a,b)]

def makeMat(nrow, ncol):
    return [[0]*ncol for i in range(nrow)]
```

- Write a function that combines every pair of elements from two lists into a list of tuples (of length 2), provided that the elements are not the same
- Example:

```
>>> magicComb([1,2,3],[3,1,4])
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

```
#Solution (without list comprehension):
def magicComb(list1, list2):
    combs = []
    for x in list1:
        for y in list2:
            if x != y:
                combs.append((x, y))
    return combs

#Solution (one-liner):
def magicComb(list1, list2):
    return [(x, y) for x in list1 for y in list2 if x != y]
```

Finger exercises

- Write a function to transpose a matrix, implemented as before (list of rows)
- Example:

```
>>> code.transpose([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

```
#Solution 0:
def transpose(mat):
    nrow = len(mat)
    ncol = len(mat[0])
    tmat = []
    for i in range(ncol):
        row = []
        for j in range(nrow):
            row.append(mat[j][i])
        tmat.append(row)
    return tmat

#Solution 1:
def transpose(mat):
    tmat = []
    for i in range(len(mat[0])):
        tmat.append([row[i] for row in matrix])
    return tmat


#Solution 2:
#shorter but not necessarily better than solution 1, it is not very readable
def transpose(mat):
    return [[row[i] for row in mat] for i in range(len(mat[0]))]
```

Dictionaries

- Very powerful, 'pythonic' object (not only in Python)
- A dictionary contains items: item=(key, value)
- values can be read, modified, inserted and deleted efficiently using their keys
- it is easy to loop through the keys, or the items
- different types in the keys or in the values can be mixed, but lists cannot be used as keys (they are not hashable)

```
>>> emptyDict = {}
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127 #inserting an item
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack'] #reading an item
4098
>>> del tel['sape'] #deleting an item
>>> tel['guido'] = 3227 #modifying an item
>>> tel
{'jack': 4098, 'guido': 3227}
>>> tel['angelinajolie'] #key not present
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'angelinajolie'
>>> for key in tel.keys(): #same as: for key in tel:
...     print(key)
...
jack
guido
>>> for key, value in tel.items():
...     print(key, value)
...
jack 4098
guido 3227
>>> dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First', 1:18, 19.2:"that's a weird key..."} #mixed key types
>>> dict[['list', 'as','key']]=3 #lists as keys not possible
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> dict["list1"]=['list', 'as', 'value'] #lists as value perfectly fine
```

Finger exercises

- Write a function that takes as input a dictionary and returns a new dictionary where keys and values are inverted
- Example:

```
>>> invertDict({'name':'ale', 'age':'impolite to ask', 'nfingers':20})
{'ale':'name', 'impolite to ask':'age', 20:'nfingers'}
```

```
#solution 1
def invertDict(dict):
    newdict = {}
    for (k, v) in dict.items():
        newdict[v] = k
    return newdict

#solution 2
def invertDict(dict):
    newdict = {}
    for key in dict.keys():
        value = dict[key]
        newdict[value] = key
    return newdict
```

Finger exercises

- Amino acids are coded in the DNA as codons: a sequence of three bases. Each codon codes for a specific amino acid, but the same amino acid can be coded by more than one codon.
- Write a function that takes as input:
  - 1. a sequence of possible codons (represented as strings, without duplicates)
  - 2. a matching sequence of amino acids (represented as strings, possibly duplicated)
  - 3. a sequence of codons
- And that returns
  - 1. the correct translation of the given codon sequence (3.) into amino acids
- Example:

```
>>> translate(['TTT','TTC','ATT','GCG'], ('Phe','Phe','Ile', 'Ala'), ['ATT','GCG','ATT','GCG','TTT','ATT','GCG'])
['Ile', 'Ala', 'Ile', 'Ala', 'Phe', 'Ile', 'Ala']
```

```
def translate(codons, aminos, seq):
    dict = {}
    for i in range(codons):
        dict[codons[i]] = aminos[i]
    return [dict[el] for el in seq]
```

File I/O

- We are going to use only line-oriented file I/O: we read from or write to a file line by line

```
>>> infile = open("secretData.txt", 'r')  #to read,  use open with the 'r' flag
>>> outfile = open("stealData.txt", 'w')  #to write,  use open with the 'w' flag
>>> infile.readline()                     #to read a line, 'use readline()'
"I'm hungry!\n"                           #lines end always with a newline character (“\n”)
>>> outfile.readline()                    #you can't read from a file opened for writing
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
io.UnsupportedOperation: not readable
>>> line = infile.readline()
>>> print(line)                           #the trailing '\n' can be annoying
I like pizza!                             #that's why here we get an extra blank line

>>> print(line[:-1])
I like pizza!                             #we can remove it in this way
>>> line=line.strip()                     #or using strip()
>>> print(line[:-1])                      #now the last character is '!'
I like pizza
>>> for line in infile:                   #looping through lines is beautiful
...     print(line.strip())
...
But also ice cream!
Today it's not very sunny...
wazzup bro?
>>> infile.close()                        #if we want to re-read a file, we need to close it
>>> infile = open("secretData.txt")       #and open it again
>>> #lines = infile.readlines()           #we can read all lines at once with readlines()
>>> for line in infile:
...     outfile.write(line)               #90% of my python scripts look like this
...
>>> infile.close()                        #better to close files at the end
>>> outfile.close()
```

Other important modules

- The random module: generating random numbers
- the re module: pattern matching with regular expressions
- matplotlib and numpy: plotting and scientific computing

The random module

```
>>> import random
>>> random.random()                          # Random float x, 0.0 <= x < 1.0
0.28537541483191564
>>> random.uniform(1, 10)                    # Random float x, 1.0 <= x < 10.0
7.777871877674695
>>> random.randrange(10)                     # Integer from 0 to 9
0
>>> random.randrange(0, 101, 2)              # Even integer from 0 to 100
14
>>> random.choice('abcdefghij')              # Choose a random element in a sequence
'h'
>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 5, 1, 3, 6, 4, 2]
>>> random.sample([1, 2, 3, 4, 5],  3)   #three samples without replacement
[2, 5, 1]
```

Finger exercises

- Write a function that takes in an alphabet as a string, a number n and returns a string of length n formed by choosing n times a random character in the alphabet.
- Example:

```
>>> rndString('acgt', 10)
gactggtcag
```

```
import random
#Solution 1:
def rndString(alpha, n):
    res = ""
    for i in range(n):
        res += random.choice(alpha)
    return res

#Solution 2:
def rndString(alpha, n):
    res = [random.choice(alpha) for i in range(n)]
    return "".join(res)
```

Pattern matching with exact patterns

- when we look for exact patterns, there is no need to use the re module, strings already have basic pattern matching capabilities

```
>>> infile = open("secretData.txt", 'r')  #to read,  use open with the 'r' flag
>>> lines = infile.readlines()            #to read all lines at once, 'use readlines()': it makes a list of lines
["I'm hungry!\n", 'I like pizza!\n', 'But also ice cream!\n', "Today it's not very sunny...\n", 'wazzup bro?\n']
>>> text = “”.join(lines)                 #str.join() joins all the element in the lists separating them with str
text
"I'm hungry!\nI like pizza!\nBut also ice cream!\nToday it's not very sunny...\nwazzup bro?\n"
>>> text.find("sistah")                   #does the pattern 'sistah' occur in the text?
-1                                        #no
>>> text.find("bro")                      #does a bro occur in the text?
82                                        #yes, at index 82
>>> text[82:(82 + len(“bro”))]            #from the index, we can slice off our occurrence from the text
'bro'
>>> text.find("I")                        #look for pattern “I”
0
>>> text.find("I", 10)                    #look for pattern “I” starting from position 10 (included)
12
>>> text.count("I")                       #count occurrences of “I”
2
>>> text.endswith("bro")                  #check pattern at the end of the text (useful for file extensions)
False
>>> text.endswith("bro?\n")
True
>>> text.startswith("I\n")                #check pattern at the beginning
False
>>> text.startswith("I am")
False
>>> text.startswith("I'm")
True
>>> text.replace("I", "you")              #replace all occurrences of a pattern with another pattern
"you'm hungry!\nyou like pizza!\nBut also ice cream!\nToday it's not very sunny...\nwazzup bro?\n"
```

Advanced pattern matching: regular expressions

- The re module handles Perl-style regular expressions (same syntax as in Perl, Unix and many other softwares)
- A good tutorial: https://docs.python.org/3.2/howto/regex.html
- We will focus on Python functions rather than on regular expression syntax

```
>>> import re                                  #module for regular expressions
>>> text = "that's an ode to the code that I wrote from remote"
>>> pattern = "o.e"                           #special character '.' means: any character
>>> m = re.match(pattern, text)               #re.match tests if the entire text matches
>>> print(m)                                  #it doesn't
None
>>> m = re.search(pattern, text)              #re.search looks for the first match in the text
>>> print(m)                                  #match object
<_sre.SRE_Match object at 0x7f864d40b510>
>>> m.start()                                 #match start
10
>>> m.end()                                   #match end
13
>>> m.group()                                 #match content
'ode
>>> re.findall(pattern, text)                 #find all the match contents
['ode', 'ode', 'ote', 'ote']
>>> miter = re.finditer(pattern, text)        #sequence of all match objects
>>> ms = list(miter)                          #convert it to a list
>>> ms
[<_sre.SRE_Match object at 0x7f864d40b510>,
 <_sre.SRE_Match object at 0x7f864d40b5e0>,
 <_sre.SRE_Match object at 0x7f864d40b648>,
 <_sre.SRE_Match object at 0x7f864d40b6b0>]
>>> [m.start() for m in ms]                   #get all starts
[10, 22, 35, 47]
>>> [m.end() for m in ms]                     #get all ends
[13, 25, 38, 50]
>>> [m.group() for m in ms]                   #get all contents
['ode', 'ode', 'ote', 'ote']
```

Advanced pattern matching: regular expressions

- regular expressions can represent also variable length matches
- with round brackets we can easily get subsets of the match
- more examples here: http://www.tutorialspoint.com/python/python_reg_expressions.htm

```
>>> text = "dogs are smarter than cats"     #* means repeat the previous pattern any number of times
>>> pattern = r"([a-z]*) are (\w+) .*"      #+ means repeat the previous pattern at least once
>>> m = re.match(pattern, text)             #[c-e] means all characters between c and e, same as [cde]
>>> m.group()                               # \w is a shortcut for [a-zA-Z]
'dogs are smarter than cats'
>>> m.group(1)                              #first round bracket
'dogs'
>>> m.group(2)                              #second round bracket
'smarter'
>>> text = "dogs are smarter than cats, her eyes are blue, they are very good students"
>>> pattern = r"([^ ]*) are (\S*)"          #[^abc] means any character ecxept a, b and c
>>> ms = list(re.finditer(pattern, text))   #\S is a shortcut for [^  \t\n\r\f\v]
>>> [m.group() for m in ms]
['dogs are smarter', 'eyes are blue,', 'they are very']
>>> [m.group(1) for m in ms]
['dogs', 'eyes', 'they']
>>> [m.group(2) for m in ms]
['smarter', 'blue,', 'very']
```

Problem: regular expressions

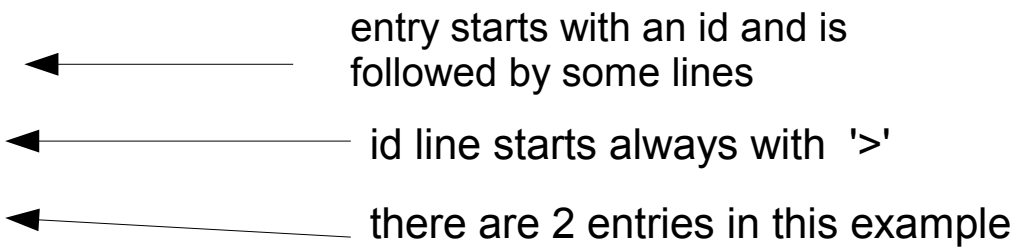
- Given a string 'text' and a character 'c', determine the lengths of all portions of the text where a 'c' appears
- Example:

```
>>> runlengths('aaaaccagaataaataaaa', 'a')
[4,1,2,3,4]
```

Problem: file I/O, read fasta

- Read a file in fasta format. It's more or less like this:

```
>id1
line
line
>id2
line
line
line
```




```
>gi|31563518|ref|NP_852610.1| microtubule-associated proteins 1A/1B light chain 3A isoform b [Homo sapiens]
MKMRFFSSPCGKAAVDPADRCKEVQQIRDQHPSKIPVIIERYKGEKQLPVLDKTKFLVPDHVNMSELVKI
IRRRLQLNPTQAFFLLVNQHSMVSVSTPIADIYEQEKDEDGFLYMVYASQETFGF
```

- input: a path to a fasta file
- output: a dictionary with the ids as keys and the sequences as values

```
>>> readFasta('path/to/my/fasta.fasta')
{'id1':'lineline', 'id2':'linelineline'}
```

for real applications, use the Biopython module for python

Problem: file I/O read bed

- Read a file in bed format. It's more or less like this:
  - fields separated by tabs (character '\t')
  - we will care about only the first 3 fields: chromosome, start and end
  - discard lines starting with “#”

```
#discard me
chromosome1     start1     end1  name1      score 1     strand1    other-stuff-we-can-ignore1       ignore..
chromosome2     start 2    end2  name2      score2      strand2    other-stuff-we-can-ignore2       ignore..
```

```
chr21      1000  5000  cloneA 960 + 1000 5000 0 2 567,488, 0,3512
chr22      2000  6000  cloneB 900 - 2000 6000 0 2 433,399, 0,3601
```

- input: a path to a bed file
- output: a list of tuples ('chr', start, end)

```
>>> readBed('path/to/my/bed.bed')
[('chr21', 1000, 5000), ('chr22', 2000, 6000)]
```

for real applications, use the Biopython module for python

Problem: file I/O, write fasta

- Read a file in fasta format and a file in bed format, for each tuple ('chr', start, end):
  - slice the sequence with id 'chr' in the fasta file from position start to position end
  - make a new id of the form: id-start-end
  - write this new sequence to an output file in fasta format

```
chromosome1     start1     end1  name1      score 1     strand1    other-stuff-we-can-ignore1       ignore..
chromosome2     start 2    end2  name2      score2      strand2    other-stuff-we-can-ignore2       ignore..
```

```
chr21      1000  5000  cloneA 960 + 1000 5000 0 2 567,488, 0,3512
chr22      2000  6000  cloneB 900 - 2000 6000 0 2 433,399, 0,3601
```

- input:
  - a path to a fasta file
  - a path to a bed file
  - an output path
- output: write in the given file (don't return anything)

```
>>> getFasta('path/to/my/fasta.fasta', 'path/to/my/bed.bed', 'output/path.fasta')
```

for real applications, use the “getfasta” utility in samtools: http://samtools.sourceforge.net/