同济大学 TONGJI UNIVERSITY

生物信息学系 DEPARTMENT OF BIOINFORMATICS

Section 7

# Python: Pattern Matching and Text Mining Divide a Program into Functions

Yong Zhang, Ph. D

School of Life Science and Technology
Tongji University

Apr 18, 2019

yzhang@tongji.edu.cn

## Story: Search a phosphorylation motif in a protein sequence

```
import re

seq = 'VSVLTMFRYAGWLDRLYMLVGTQLAAIIHGVALPLMMLI'

pattern = re.compile('[ST]Q')

match = pattern.search(seq)
if match:
    print '%10s' %(seq[match.start() - 4:match.end() + 4])
    print '%6s' % match.group()
else:
    print "no match"
```

## Compiling regular expressions

```
import re

seq = 'VSVLTMFRYAGWLDRLYMLVGTQLAAIIHGVALPLMMLI'

pattern = re.compile('[ST]Q')

match = pattern.search(seq)
if match:
    print '%10s' %(seq[match.start() - 4:match.end() + 4])
    print '%6s' % match.group()
else:
    print "no match"
```

- *compile()* is the method to compile a string and convert it into a regular expression object (the *RegexpObject*).

## Pattern matching

```
import re

seq = 'VSVLTMFRYAGWLDRLYMLVGTQLAAIIHGVALPLMMLI'

pattern = re.compile('[ST]Q')

match = pattern.search(seq)
if match:
    print '%10s' %(seq[match.start() - 4:match.end() + 4])
    print '%6s' % match.group()
else:
    print "no match"
```

## Pattern matching

- Once your regular expression is compiled, and you have a *RegexpObject*, you can search for its matches in a string using *RegexpObject* methods.
- The *search()* method scans a string, looking for a location where the regular expression matches for the first time.

```
>>> import re
>>> motif = 'R.[ST][^P]'
>>> regexp = re.compile(motif)
>>> seq = 'RQSAMGSNKSKPKDASQRRRSLEPAENVHGAGGGAFPASQRPSKP'
>>> match = regexp.search(seq)
>>> match
<_sre.SRE_Match object at 0x100650780>
```

## Pattern matching

- the *search()* method returns not the matching substring directly but a *Match* object, which encodes the matching substring and its start and end positions along the sequence. This information can be retrieved using the following methods of a match object:

```
>>> match.group()  #returns the matching substring
'RQSA'
>>> match.span()   #returns a tuple of start and end positions of the match
(0, 4)
>>> match.start()  #returns the start position of the match
0
>>> match.end()    #returns the end position of the match
4
```

## Pattern matching

- If you are interested in finding the regular expression match starting at the first position of a sequence, you can use the method *match()*.

```
>>> seq = 'RQSAMGSNKSKPKDASQRRRSLEPAENVHGAGGGAFPASQRPSKP'
>>> motif = 'R.[ST][^P]'
>>> regexp = re.compile(motif)
>>> match1 = regexp.match(seq)
>>> match1.group()
'RQSA'
>>> match1.span()
(0, 4)
>>> match1.start()
0
>>> match1.end()
4
```

## How to find all matches?

- The *findall()* method returns a list containing all the matching substrings.

```
>>> import re
>>> seq = 'RQSAMGSNKSKPKDASQRRRSLEPAENVHGAGGGAFPASQRPSKP'
>>> motif = 'R.[ST][^P]'
>>> regexp = re.compile(motif)
>>> all = regexp.findall(seq)
>>> all
['RQSA', 'RRSL', 'RPSK']
```

- The *finditer()* method finds all the *Match* objects corresponding to the regular expression matches.

```
>>> iter = regexp.finditer(seq)
>>> for s in iter:
...     print s.group()
...
RQSA
RRSL
RPSK
```

## Grouping

- The *group()* method with no argument (or the argument equal to 0) always returns the complete matching substring, whereas subgroups are numbered from left to right in increasing order (starting with 1).

```
>>> import re
>>> seq = 'QSAMGSNKSKPKDASQRRRSLEPAENVHGAGGGAFPASQRPSKP'
>>> pattern1 = re.compile('R(.)[ST][^P]')
>>> match1 = pattern1.search(seq)
>>> print match1.group()
RRSL
>>> print match1.group(1)
R
>>> pattern2 = re.compile('R(.{0,3})[ST][^P]')
>>> match2 = pattern2.search(seq)
>>> print match2.group()
RRRSL
>>> print match2.group(1)
RR
```

## Grouping

- Subgroups could be nested, and to know the corresponding number, you have to count the number of open round brackets from left to right.
- The *groups()* method returns a tuple with the substrings corresponding to all subgroups.

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
>>> m.group(2,1,2)
('b', 'abc', 'b')
>>> m.group()
'abcd'
>>> m.groups()
('abc', 'b')
```

## Grouping

- You can assign a name (label) to each subgroup in order to selectively retrieve its content.
- The group label must be put between < and > symbols and inserted in the round brackets of the group preceded by *?P*.

```
>>> pattern = 'R(?P<w1>.{0,3})[ST](?P<w2>[^P])'
>>> regexp = re.compile(pattern)
>>> m1 = regexp.search(seq)
>>> m1.group()
'RRRSL'
>>> m1.group('w1')
'RR'
>>> m1.group(1)
'RR'
>>> m1.group('w2')
'L'
>>> m1.group(2)
'L'
```

## Modifying strings

- The *re* module provides three methods that allow modifying strings: *split(s)*, *sub(r, s, [c])*, and *subn(r, s, [c])*.
- The method *split(s)* splits the string s at the matches of a regular expression.

```
>>> import re
>>> separator = re.compile('\|')  # a "\" before the metacharacter "|"
>>> annotation = 'ATOM:CA|RES:ALA|CHAIN:B|NUMRES:166'
>>> columns = separator.split(annotation)
>>> print columns
['ATOM:CA', 'RES:ALA', 'CHAIN:B', 'NUMRES:166']
```

## Modifying strings

- The *RegexpObject* method *sub(r, s, [c])* returns a new string where nonoverlapping occurrences of a given pattern in the *s* string are all replaced with the value of *r* (if the optional argument *c* is not specified).
- The optional argument *c* is the maximum number of pattern occurrences to be replaced.

```
>>> import re
>>> separator = re.compile('\|')
>>> annotation = 'ATOM:CA|RES:ALA|CHAIN:B|NUMRES:166'
>>> new_annotation = separator.sub('@', annotation)
>>> print new_annotation
ATOM:CA@RES:ALA@CHAIN:B@NUMRES:166
>>> new_annotation = separator.sub('@', annotation, 2)
>>> print new_annotation
ATOM:CA@RES:ALA@CHAIN:B|NUMRES:166
```

## Modifying strings

- The method *subn(r, s, [c])* does the same but returns a tuple of two elements, where the first element is the new string and the second is the number of replacements that were performed.

```
>>> new_annotation = separator.subn('@', annotation)
>>> print new_annotation
('ATOM:CA@RES:ALA@CHAIN:B@NUMRES:166', 3)
```

## Metacharacters

- The regular expression metacharacters are *[ ] ^ $ \ . | * + ? { } ( )*
- *[ ]*
  - Square brackets are used to indicate a class of characters.
  - If you search a match of *[abc]* in a string *s*, you will find it if *s* contains '*a*', '*b*', or '*c*'.
  - *[a-z]* matches the class of alphabet characters from *a* to *z*, whereas *[0-9]* matches the integers between *0* and *9*.
- ^
  - *[^a]* indicates the complement of *a*, i.e., every character different from *a*.
  - *^a* (not enclosed in square brackets) indicates that a match exists in s only if *a* is in the first position of *s*.

## Metacharacters

- \
  - The meaning of \ depends on whether \ is followed by a metacharacter or a character. In the first case, it "protects" the metacharacter by restoring its literal meaning; in the second case, its meaning depends on the character that follows.
    - *d* corresponds to *[0-9]*;
    - *D* corresponds to *[^0-9]*;
    - *s* corresponds to *[\t\n\r\f\v]*, i.e., any whitespace character
    - *S* corresponds to *[^\t\n\r\f\v]*, i.e., any character that is not a whitespace
    - *w* corresponds to *[a-zA-Z0-9]*, i.e., any alphanumeric character
    - *W* corresponds to *[^a-zA-Z0-9]*, i.e., any character that is not alphanumeric.

## Metacharacters

- *$*
  - *a$* indicates that a match exists in *s* only if *a* is in the last position of *s*.
- .
  - This corresponds to any character except the newline character.
- |
  - This is the OR operator. If placed between two regular expressions, matches will be searched either with the regexp on its left or with the one on its right.
- *( )*
  - Round brackets are used to create subgroups in a regular expression.

## Metacharacters

- Repetitions: ** + ? { }*
  - These metacharacters are used to find a match with repeated things.
  - *** The preceding character can be matched zero or more times. *a*bc* will match "*bc*", "*abc*", "*aabc*", "*aaabc*" etc.
  - *+* The preceding character can be matched one or more times. *a+bc* will match "*abc*", "*aabc*", "*aaabc*" etc. but not "*bc*."
  - *?* The preceding character can be matched zero times or once. *can-?can* will match both "*can-can*" and "*cancan*."
  - *{m,n}* This qualifier means that at least *m* and at most *n* repetitions of the preceding character will be matched.

## Example

- How to find transcription factor binding sites in a genomic sequence?
  - The '*TFBS.txt*' file contains this list of TFBSs:

```
UAS(G)-pMH100 CGGAGTACTGTCCTCCG ! J Mol Biol 209: 423-32 (1989)
TFIIIC-X1s-50 TGGATGGGAG ! EMBO J 6: 3057-63 (1987)
HSE_CS_inver0 CTNGAANNTTCNAG ! Cell 30: 517-28 (1982)
ZDNA_CS 0 GCGTGTGCA ! Nature 303: 674-9 (1983)
GCN4-his3-180 ATGACTCAT ! Science 234: 451-7 (1986)
```

  - The '*genome.txt*' file contains the sequence in FASTA format a whole chromosome of a eukaryotic organism.

## Example

```
import re

genome_seq = open('genome.txt').read()

# read transcription factor binding site patterns
sites = []
for line in open('TFBS.txt'):
    fields = line.split()
    tf = fields[0]
    site = fields[1]
    sites.append((tf, site))

# match all TF's to the genome and print matches
for tf, site in sites:
    tfbs_regexp = re.compile(site)
    all_matches = tfbs_regexp.findall(genome_seq)
    matches = tfbs_regexp.finditer(genome_seq)
    if all_matches:
        print tf, ':'
        for tfbs in matches:
            print '\t', tfbs.group(), tfbs.start(), tfbs.end()
```

## Example

- Extract the title and the abstract text from a PubMed HTML page.

```
import urllib2
import re

pmid = '18235848'
url = 'http://www.ncbi.nlm.nih.gov/pubmed?term=%s' % pmid
handler = urllib2.urlopen(url)
html = handler.read()

title_regexp = re.compile('<h1>(.{5,400})</h1>')
title_text = title_regexp.search(html)
abstract_regexp = re.compile('<h3>Abstract</h3><div class=""><p>(.*)</p></div>')
abstract_text = abstract_regexp.search(html)

print 'TITLE:', title_text.group(1)
print 'ABSTRACT:', abstract_text.group(1)
```

## Example

- The *urllib2* module provides tools to connect to a URL and retrieve its content.
- The *urlopen()* is the method for URL opening, and it returns a file-type Python object (a *handler*).
- The *read()* method reads the handler content as a single string of text.

## Example

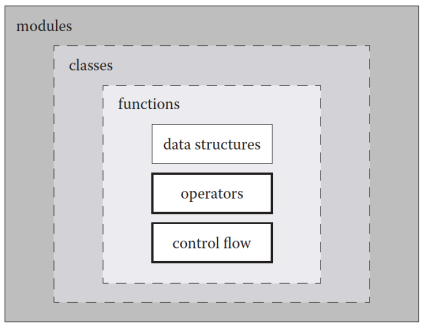
- Detect a specific word or a set of words in a scientific abstract.

```
import urllib2
import re

# word to be searched
word_regexp = re.compile('schistosome')

# list of PMIDs where we want to search the word
pmids = ['18235848', '22607149', '22405002', '21630672']
for pmid in pmids:
    url = 'http://www.ncbi.nlm.nih.gov/pubmed?term=' + pmid
    handler = urllib2.urlopen(url)
    html = handler.read()
    title_regexp = re.compile('<h1>(.{5,400})</h1>')
    title = title_regexp.search(html)
    title = title.group(1)
    abstract_regexp = re.compile('<h3>Abstract</h3><div class=""><p>(.*)</p></div>')
    abstract = abstract_regexp.search(html)
    abstract = abstract.group(1)
    all_words = word_regexp.findall(abstract, re.IGNORECASE)
    words = word_regexp.finditer(abstract, re.IGNORECASE)
    if all_words:
        # display title and where the keyword was found
        print title
        for word in words:
            print word.group(), word.start(), word.end()
```

## The Python structure



## Story: Working with 3D coordinate files

- PDB files are text files that contain both annotation and atomic coordinates (*x,y,z*) of biological molecules.
  - The first part of the record, called the *header*, includes several annotation lines.
  - The second part of the record reports the atomic coordinate lines for standard groups. They start with the "*ATOM*" keyword and are separated into columns.
    - The columns format can be translated into a Python string: *pdb_format = '6s5s1s4s1s3s1s1s4s1s3s8s8s8s6s6s6s4s2s3s'*
    - *6s* corresponds to columns 1-6, *5s* to columns 7-11, etc.

## Example Python session

```
import struct

pdb_format = '6s5s1s4s1s3s1s1s4s1s3s8s8s8s6s6s10s2s3s'

amino_acids = {
    'ALA':'A', 'CYS':'C', 'ASP':'D', 'GLU':'E',
    'PHE':'F', 'GLY':'G', 'HIS':'H', 'LYS':'K',
    'ILE':'I', 'LEU':'L', 'MET':'M', 'ASN':'N',
    'PRO':'P', 'GLN':'Q', 'ARG':'R', 'SER':'S',
    'THR':'T', 'VAL':'V', 'TYR':'Y', 'TRP':'W'
    }

def threeletter2oneletter(residues):
    '''
    Converts the first element of each residue
    to a one letter amino acid symbol
    '''
    for i, threeletter in enumerate(residues):
        residues[i][0] = amino_acids[threeletter[0]]
```

## Example Python session

```
def get_residues(pdb_file):
    '''
    Reads the PDB input file, extracts the
    residue type and chain from the CA lines
    and appends both to the residues list
    '''
    residues = []
    for line in pdb_file:
        if line[0:4] == "ATOM":
            tmp = struct.unpack(pdb_format, line)
            ca = tmp[3].strip()
            if ca == 'CA':
                res_type = tmp[5].strip()
                chain = tmp[7]
                residues.append([res_type, chain])
    return residues
```

## Example Python session

```
def write_fasta_records(residues, pdb_id, fasta_file):
    '''
    Write a FASTA record for each PDB chain
    '''
    seq = ''
    chain = residues[0][1]
    for aa, new_chain in residues:
        if new_chain == chain:
            seq = seq + aa
        else:
            # write sequence in FASTA format
            fasta_file.write(">%s_%s\n%s\n" % (pdb_id, chain, seq))
            seq = aa
            chain = new_chain
    # write the last PDB chain
    fasta_file.write(">%s_%s\n%s\n" % (pdb_id, chain, seq))
```

## Example Python session

```
def extract_sequence(pdb_id):
    '''
    Main function: Opens files, writes files
    and calls other functions.
    '''
    pdb_file = open(pdb_id + ".pdb")
    fasta_file = open(pdb_id + ".fasta", "w")
    residues = get_residues(pdb_file)
    threeletter2oneletter(residues)
    write_fasta_records(residues, pdb_id, fasta_file)
    pdb_file.close()
    fasta_file.close()

# call the main function
extract_sequence("3G5U")
```

## How to define and call a function

- The instructions to define a new function are as follows:

```
def my_function(arg1, arg2,…):
        '''documentation'''
        <instructions>
        return value1, value2, …
```

  - *my_function*: the function name
  - *(arg1, arg2, …)*: the arguments, optional
  - *documentation*: the triple-quoted description, optional
  - *<instructions>*: the instructions that are executed when the function is called
  - *return*: the instruction that makes the interpreter stop executing further instructions in the function and go back to the program line from which the function was called.
  - *value1, value2, …*: the values returned by the function

## How to define and call a function

```
def calc_sum(num1, num2):
    '''here you are defining the calc_sum function'''
    result = num1 + num2
    return result

# here you are calling calc_sum
print calc_sum(12, 8)
```

## Function arguments

- Function arguments are a way to pass data to a function.
- Almost every Python object can be passed as an argument to a function. The result of a function call can be the argument of a function.

```
def increment(number):
    '''returns the given number plus one'''
    return number + 1

def print_arg(number):
    '''prints the argument'''
    print number

print_arg(increment(5))
```

## Function arguments

- There are four kinds of arguments in Python: required arguments, keyword arguments, default arguments, and variable-length arguments.
- *Required arguments:* One or multiple parameters must be passed to a function. The order of the arguments in the call must be exactly the same as the order in the function definition:

```
def print_funct(num, seq):
    print num, seq

print_funct(10, "ACCTGGCACAA")
```

## Function arguments

- *Keyword arguments*: It is possible to assign a name to the arguments of a function. In this case, the order is not important:

```
def print_funct(num, seq):
    print num, seq

print_funct(seq = "ACCTGGCACAA", num = 10)
```

- *Default arguments:* It is also possible to use default (optional) arguments. These optional arguments must be placed in the last position(s) of the function definition:

```
def print_funct(num, seq = "A"):
    print num, seq

print_funct(10, "ACCTGGCACAA")
print_funct(10)
```

## Function arguments

- *Variable-length arguments:* The number of arguments can be variable (i.e., changing from one function call to the other); they are indicated by the symbol * (for a tuple) or a ** (for a dictionary):

```
>>> def print_args(*args):
...     print args
...
>>> print_args(1,2,3,4,5)
(1, 2, 3, 4, 5)
>>> print_args("Hello world!")
('Hello world!',)
>>> print_args(100, 200, "ACCTGGCACAA")
(100, 200, 'ACCTGGCACAA')
>>> def print_args2(**args):
...     print args
...
>>> print_args2(num = 100, num2 = 200, seq = "ACCTGGCACAA")
{'num': 100, 'seq': 'ACCTGGCACAA', 'num2': 200}
```

## The *struct* module

- The *struct* module provides methods that make it possible to convert a string into a tuple on the basis of a customized format, or vice versa.
- The *struct* method *pack(format, v1, v2, …)* returns a single string made up of the *v1, v2, …* values packed according to the *format* string.

```
>>> import struct
>>> format = '2s1s1s1s1s'
>>> a = struct.pack(format,'10','2','3','4','5')
>>> a
'102345'
>>> b = struct.pack(format,'1','2','3','4','5')
>>> b
'1\x002345'
>>> c = struct.pack(format,'10','20','3','4','5')
>>> c
'102345'
```

## The *struct* module

- The method *unpack(format, string)* unpacks a string into a tuple, according to the format encoded by *format*. The string must contain the same number of characters present in the format string.

```
>>> import struct
>>> format = '1s2s1s1s'
>>> line = '12345'
>>> col = struct.unpack(format, line)
>>> col
('1', '23', '4', '5')
>>> line2 = '123456'
>>> col2 = struct.unpack(format, line2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
struct.error: unpack requires a string argument of length 5
```

## The *struct* module

- The *struct* method *calcsize(fmt)* returns the total number of characters of a given formatting string:

```
>>> import struct
>>> format = '30s30s20s1s'
>>> struct.calcsize(format)
81
```

## The *struct* module

```
def get_residues(pdb_file):
    '''
    Reads the PDB input file, extracts the
    residue type and chain from the CA lines
    and appends both to the residues list
    '''
    residues = []
    for line in pdb_file:
        if line[0:4] == "ATOM":
            tmp = struct.unpack(pdb_format, line)
            ca = tmp[3].strip()
            if ca == 'CA':
                res_type = tmp[5].strip()
                chain = tmp[7]
                residues.append([res_type, chain])
    return residues
```

```
>>> import struct
>>> line = 'ATOM  11422  CA  TYR B 322      10.356  43.497 110.570  1.00153.06           C   '
>>> format = '6s5s1s4s1s3s1s1s4s1s3s8s8s8s6s6s10s2s3s'
>>> col = struct.unpack(format, line)
>>> col
('ATOM  ', '11422', ' ', ' CA ', ' ', 'TYR', ' ', 'B', ' 322', ' ', '   ', '  10.356', '  43.497'
, ' 110.570', '  1.00', '153.06', '          ', ' C', '   ')
>>> col[5]
'TYR'
>>> col[7]
'B'
```

## Loops with *range()* and *xrange()*

- *range(n, m)* creates a list of integers ranging from *n* to *m-1*. If *n* is omitted, 0 is used as a default value.
- *xrange(n,m)* creates an iterator.
  - It does not return a list but yields the same values as the corresponding list only at the time they are needed.
  - *xrange()* consumes less memory and can be used with big numbers.
- Both methods make it possible to specify a step:
  - *range([start], stop, [step])*

## Multiple function returned values are tuples

```
>>> def f(a, b):
...     return a + b, a * b, a - b
...
>>> result = f(10,15)
>>> result
(25, 150, -5)
>>> sum, prod, diff = f(20, 2)
>>> sum
22
>>> prod
40
>>> diff
18
```

## Ten things about Python functions

1. The statement to define a function is *def*.
2. A function must be defined and called using round brackets.
3. The body of a function is a block of code that is initiated by a colon character followed by indented instructions.
4. The last indented statement marks the end of a function definition.
5. You can pass arguments to a function. Multiple arguments have the form of a tuple.
6. You can define variables in the body of a function.

## Ten things about Python functions

7. The statement *return* exits a function, optionally passing back a value to the caller. Multiple values have the form of a tuple.
8. A *return* statement with no values is possible, as well as a function with no *return* statement. In both cases the default returned value is *None*.
9. You can insert a documentation string in quotation marks in the body of a function. This string is ignored upon function call but can be retrieved using the __*doc*__ attribute of the function object.

## Ten things about Python functions

10. When a function is called, a local namespace is automatically created. The variables defined in the body of a function live in its *local* namespace and not in the *global* namespace of the script or module. When a function is called, names of the objects used in its body are first searched in the function namespace, and subsequently, if they are not found in the function body, they are searched in the global namespace of the script or module.

## *lambda* functions

- *lambda* statement can create small anonymous functions:
  - They are called *anonymous* because they are not declared in the standard manner by using the *def* statement.
  - *lambda* functions are particularly useful when they are used as arguments of other functions.
  - *lambda* functions do not contain a *return* statement: they contain an expression, the value of which is always returned.

```
>>> def f(x): return x**2
...
>>> print f(8)
64
>>> g = lambda x: x**2
>>> print g(8)
64
>>> (lambda x: x**2)(3)
9
```

## Examples

- How to write a function that calculates the distance between two points in Cartesian space?

```
from math import sqrt


def calc_dist(p1, p2):
    '''
    Returns the distance between two 3D points.
    '''
    dx = p1[0] - p2[0]
    dy = p1[1] - p2[1]
    dz = p1[2] - p2[2]
    distsq = pow(dx, 2) + pow(dy, 2) + pow(dz, 2)
    distance = sqrt(distsq)
    return distance
```

## Examples

- How to write a function that takes as input any number of arguments and returns a string of tab-separated arguments ending with a newline character?

```
def convert_to_string(*args):
    '''returns all arguments as a single tab-separated string'''
    result = [str(a) for a in args]
    return '\t'.join(result) + '\n'


output_file = open("nucleotideSubstitMatrix.txt", "w")
output_file.write(convert_to_string('', 'A', 'T', 'C', 'G'))
output_file.write(convert_to_string('A', 1.0))
output_file.write(convert_to_string('T', 0.5, 1.0))
output_file.write(convert_to_string('C', 0.1, 0.1, 1.0))
output_file.write(convert_to_string('G', 0.1, 0.1, 0.5, 1.0))
output_file.close()
```

## Examples

- How to calculate the distance between two atoms in a PDB chain?
  - parse_pdb.py

```
import struct

pdb_format = '6s5s1s4s1s3s1s1s4s1s3s8s8s8s6s6s10s2s3s'

def parse_atom_line(line):
    '''Returns an ATOM line parsed to a tuple '''
    tmp = struct.unpack(pdb_format, line)
    atom = tmp[3].strip()
    res_type = tmp[5].strip()
    res_num = tmp[8].strip()
    chain = tmp[7].strip()
    x = float(tmp[11].strip())
    y = float(tmp[12].strip())
    z = float(tmp[13].strip())
    return chain, res_type, res_num, atom, x, y, z
```

## Examples

```
from math import sqrt
from distance import calc_dist
from parse_pdb import parse_atom_line

pdb = open('3G5U.pdb')
points = []

while len(points) < 2:
    line = pdb.readline()
    if line.startswith("ATOM"):
        chain, res_type, res_num, atom, x, y, z = parse_atom_line(line)
        if res_num == '123' and chain == 'A' and atom == 'CA':
            points.append((x, y, z))
        if res_num == '209' and chain == 'A' and atom == 'CA':
            points.append((x, y, z))

print calc_dist(points[0], points[1])
```

## Examples

- How to calculate the distance between all CA atoms in a PDB chain?

```
from math import sqrt
from distance import calc_dist
from parse_pdb import parse_atom_line

def get_ca_atoms(pdb_filename):
    '''returns a list of all C-alpha atoms in chain A'''
    pdb_file = open(pdb_filename, "r")
    ca_list = []
    for line in pdb_file:
        if line.startswith('ATOM'):
            data = parse_atom_line(line)
            chain, res_type, res_num, atom, x, y, z = data
            if atom == 'CA' and chain == 'A':
                ca_list.append(data)
    pdb_file.close()
    return ca_list
```

## Examples

```
ca_atoms = get_ca_atoms("3G5U.pdb")
for i, atom1 in enumerate(ca_atoms):
    # save coordinates in a variable
    name1 = atom1[1] + atom1[2]
    coord1 = atom1[4:]
    # compare atom1 with all other atoms
    for j in range(i+1, len(ca_atoms)):
        atom2 = ca_atoms[j]
        name2 = atom2[1] + atom2[2]
        coord2 = atom2[4:]
        # calculate the distance between atoms
        dist = calc_dist(coord1, coord2)
        print name1, name2, dist
```

## Summary

- Managing Your Biological Data with Python
  - Chapter 9. Pattern Matching and Text Mining
  - Chapter 10. Divide a Program into Functions
- Python codes in https://bitbucket.org/krother/python-for-biologists/src/