Python is an easy to learn, powerful programming language

• It is simple to use: Python syntax is clearly deTined which makes it easily readable. • It is interactive: you can write and test your programs directly from a terminal window.

• It has a large standard library: Python's library of built-in functions offers a wide range of programs that are already written for you.

• It is portable: Python runs on many Unix variants, on Mac OS, and on Windows.

• It is extensible: you can use it as an extension language for applications that need a programmable interface.

• It is scalable: you can use it for very small or very large programs.

PYTHON FOR GENOMIC DATA SCIENCE

let's start!

1. NUMBERS AND STRINGS

Using Python As A Calculator

In [ ]:
```
print("hello")
```

```
hello
```

In [ ]:
```
5+5
```

Out[ ]:
```
10
```

In [ ]:
```
#** is used to calculate powers
10**2
```

Out[ ]:
```
100
```

In [ ]:
```
#* takes precedence over +, -
10.5-2*3
```

Out[ ]:
```
4.5
```

In [ ]:
```
#the % operator returns the remainder after division
17 % 3
```

Out[ ]:
```
2
```

In [ ]:
```
5 * 3 + 2
```

Out[ ]:
```
17
```

In [ ]:
```
#floor division discards the fractional part
17.0 // 3
```

Out[ ]:
```
5.0
```

Strings

• Single quoted strings:
'atg'

• Double quoted strings:
"atg"

In [ ]:
```
# + concatenate
'atg' + 'gtacgtccgt'
```

Out[ ]:
```
'atggtacgtccgt'
```

In [ ]:
```
#* copy string  (replicate)
'atg'*3
```

Out[ ]:
```
'atgatgatg'
```

In [ ]:
```
#in     : membership:   true    if      Wirst   string  exists  inside
'atg' in 'atggccggcgta'
```

Out[ ]:
```
True
```

In [ ]:
```
#not in: non-membership:        true    if      Wirst   string  does    not      exi
'n' not in 'atgtgggg'
```

Out[ ]:
```
True
```

2. VARIABLES

myvariable = 1 , MyVariable = 2 , MYVARIABLE = 3

Valid names: name, _str, DNA , sequence1

Invalid names: 1string, name#, year@20

In [ ]:
```
# assigning a variable
dna="gatcccccgatattatttgc"
dna
```

Out[ ]:
```
'gatcccccgatattatttgc'
```

In [ ]:
```
#trying indexing
dna[0]
```

Out[ ]:
```
'g'
```

In [ ]:
```
dna[-3]
```

Out[ ]:
```
't'
```

In [ ]:
```
dna[3:]
```

Out[ ]:
```
'cccccgatattatttgc'
```

In [ ]:
```
#finding length
len(dna)
```

Out[ ]:
```
20
```

In [ ]:
```
#value count
dna.count('c')
```

Out[ ]:
```
6
```

In [ ]:
```
dna.count('gc')
```

Out[ ]:
```
1
```

In [ ]:
```
#upper case
dna.upper()
```

Out[ ]:
```
'GATCCCCCGATATTATTTGC'
```

In [ ]:
```
dna
```

Out[ ]:
```
'gatcccccgatattatttgc'
```

In [ ]:
```
dna.find('at',9)
```

Out[ ]:
```
9
```

In [ ]:
```
#finding
dna.find('at')
```

Out[ ]:
```
1
```

In [ ]:
```
#finding starting from reverse
dna.rfind('at')
```

```
14
```

Out[ ]:

In [ ]:
```
# true if it is lower
dna.islower()
```

Out[ ]:
```
True
```

In [ ]:
```
# true if it is upper
dna.isupper()
```

Out[ ]:
```
False
```

In [ ]:
```
# replace first arg present in var, with second arg
dna.replace('a','A')
```

Out[ ]:
```
'gAtcccccgAtAttAtttgc'
```

In [ ]:
```
#storing in variables
no_g=dna.count('g')
no_c=dna.count('c')
dna_length=len(dna)
```

In [ ]:
```
#compute the GC%
gc_percent=(no_c+no_g)*100.0/dna_length
gc_percent
```

Out[ ]:
```
45.0
```

In [ ]:
```
#Fancier        Output  Formatting
print("The DNA sequence's GC content is",gc_percent,'%')
```

```
The DNA sequence's GC content is 45.0 %
```

In [ ]:
```
#input function
dna1=input("Enter a DNA sequence, please:")
```

In [ ]:
```
dna1
```

Out[ ]:
```
'TCAGTCCATCGAGAAGAAGTACTTCGATTTCGACATCTACTGATCGTCGGG'
```

In [ ]:
```
# find the type of var
type(dna1)
```

Out[ ]:
```
str
```

In [ ]:
```
#converts       an      integer to      a       character
chr(64)
```

Out[ ]:
```
'@'
```

In [ ]:
```
#converts       x       to      a       string
str(dna1)
```

Out[ ]:
```
'TCAGTCCATCGAGAAGAAGTACTTCGATTTCGACATCTACTGATCGTCGGG'
```

In [ ]:
```
type(dna1)
```

Out[ ]:
```
str
```

3. LISTS

A list is an ordered set of values

In [ ]:
```
# creating a list
gene_expression=['gene',5.16e-08, 0.000138511, 7.33e-08]
gene_expression
```

Out[ ]:
```
['gene', 5.16e-08, 0.000138511, 7.33e-08]
```

In [ ]:
```
#indexing
print(gene_expression[2])
```

```
0.000138511
```

In [ ]:
```
#mutate string
gene_expression[0]='Lif'
gene_expression
```

Out[ ]:
```
['Lif', 5.16e-08, 0.000138511, 7.33e-08]
```

In [ ]:
```
#Don't  change  an      element in      a       string!
motif ='nacggggtc'
motif[0]='a'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_10364/448929774.py in <module>
      1 #Don't  change  an      element in      a       string!
      2 motif ='nacggggtc'
----> 3 motif[0]='a'

TypeError: 'str' object does not support item assignment
```

In [ ]:
```
#indexing
gene_expression[-3:]
```

Out[ ]:
```
[5.16e-08, 0.000138511, 7.33e-08]
```

In [ ]:
```
#indexing
gene_expression[:]
```

Out[ ]:
```
['Lif', 5.16e-08, 0.000138511, 7.33e-08]
```

In [ ]:
```
# finding length
len(gene_expression)
```

Out[ ]:
```
4
```

In [ ]:
```
# can delete an index
del gene_expression[1]
```

In [ ]:
```
gene_expression
```

Out[ ]:
```
['Lif', 0.000138511, 7.33e-08]
```

In [ ]:
```
#adding more items at end
gene_expression.extend([5.16e-08, 0.000138511])
gene_expression
```

Out[ ]:
```
['Lif', 0.000138511, 7.33e-08, 5.16e-08, 0.000138511]
```

In [ ]:
```
# showing reverse
gene_expression.reverse()
gene_expression
```

Out[ ]:
```
[0.000138511, 5.16e-08, 7.33e-08, 0.000138511, 'Lif']
```

In [ ]:
```
#try function help for more info/detail
#help(list)
```

In [ ]:
```
#Lists  As      Stacks
stack=['a','b','c','d']
```

In [ ]:
```
stack.append('e')
stack
```

Out[ ]:
```
['a', 'b', 'c', 'd', 'e']
```

In [ ]:
```
#To     retrieve        an      item    from    the     top     of      the     stac
elem=stack.pop()
```

In [ ]:
```
#order the list
mylist=[3,31,123,1,5]
sorted(mylist) #another way     mylist.sort()
```

Out[ ]:
```
[1, 3, 5, 31, 123]
```

In [ ]:
```
#order the list
mylist=['c','g','T','a','A']
sorted(mylist)
```

Out[ ]:
```
['A', 'T', 'a', 'c', 'g']
```

4. TUPLES

A tuple consists of a number of values separated by commas, and is another standard sequence data type, like strings and lists.

In [ ]:
```
t=1,2,3
#or
t=(1,2,3)
t
```

Out[ ]:
```
(1, 2, 3)
```

5. SETS

A set is an unordered collection with no duplicate elements.
Set objects support mathematical operations like union,
intersection, and difference.

In [ ]:
```
#creating a set
brca1={'DNA repair','zinc ion binding','DNA binding','ubiquitin-protein transferase
'DNA repair','protein ubiquitination'}
brca1
```

Out[ ]:
```
{'DNA binding',
 'DNA repair',
 'protein ubiquitination',
 'ubiquitin-protein transferase activity',
 'zinc ion binding'}
```

In [ ]:
```
#creating another set
brca2={'protein binding','H4 histone acetyltransferase activity','nucleoplasm', 'DNA
'double-strand break repair via homologous recombination'}
brca2
```

Out[ ]:
```
{'DNA repair',
 'H4 histone acetyltransferase activity',
 'double-strand break repair',
 'double-strand break repair via homologous recombination',
 'nucleoplasm',
 'protein binding'}
```

In [ ]:
```
#union
brca1 | brca2
```

Out[ ]:
```
{'DNA binding',
 'DNA repair',
 'H4 histone acetyltransferase activity',
 'double-strand break repair',
 'double-strand break repair via homologous recombination',
 'nucleoplasm',
 'protein binding',
 'protein ubiquitination',
 'ubiquitin-protein transferase activity',
 'zinc ion binding'}
```

In [ ]:
```
#intersection
```

```
brca1 & brca2
```

Out[ ]:
```
{'DNA repair'}
```

In [ ]:
```
#difference
brca1 - brca2
```

Out[ ]:
```
{'DNA binding',
 'protein ubiquitination',
 'ubiquitin-protein transferase activity',
 'zinc ion binding'}
```

6. DICTIONARIES

A dictionary is an unordered set of key and value pairs, with the requirement that the keys are unique (within one dictionary).

In [ ]:
```
#creating a dictionary
TF_motif = {'SP1' :'gggcgg',
'C/EBP':'attgcgcaat',
'ATF':'tgacgtca',
'c-Myc':'cacgtg',
'Oct-1':'atgcaaat'} #Each       key     is      separated       from    its      val
```

In [ ]:
```
# can asses value by its key
print("The recognition sequence for the ATF transcription is",
TF_motif['ATF'])
```

```
The recognition sequence for the ATF transcription is tgacgtca
```

In [ ]:
```
#Add    a       new     key:value       pair    to      the     dictionary
TF_motif['AP-1']='tgagtca'
TF_motif
```

Out[ ]:
```
{'SP1': 'gggcgg',
 'C/EBP': 'attgcgcaat',
 'ATF': 'tgacgtca',
 'c-Myc': 'cacgtg',
 'Oct-1': 'atgcaaat',
 'AP-1': 'tgagtca'}
```

In [ ]:
```
#Modify an      existing        entry
TF_motif['AP-1']='tga(g/c)tca'
TF_motif
```

Out[ ]:
```
{'SP1': 'gggcgg',
 'C/EBP': 'attgcgcaat',
 'ATF': 'tgacgtca',
 'c-Myc': 'cacgtg',
 'Oct-1': 'atgcaaat',
 'AP-1': 'tga(g/c)tca'}
```

In [ ]:
```
#Delete a       key     from    the     dictionary
del TF_motif['SP1']
TF_motif
```

```
{'C/EBP': 'attgcgcaat',
```

Out[ ]:
```
'ATF': 'tgacgtca',
'c-Myc': 'cacgtg',
'Oct-1': 'atgcaaat',
'AP-1': 'tga(g/c)tca'}
```

In [ ]:
```
#find length
len(TF_motif)
```

Out[ ]:
```
5
```

In [ ]:
```
# list the keys only
list(TF_motif.keys())
```

Out[ ]:
```
['C/EBP', 'ATF', 'c-Myc', 'Oct-1', 'AP-1']
```

In [ ]:
```
# list the values only
list(TF_motif.values())
```

Out[ ]:
```
['attgcgcaat', 'tgacgtca', 'cacgtg', 'atgcaaat', 'tga(g/c)tca']
```

In [ ]:
```
#order
sorted(TF_motif.keys())
```

Out[ ]:
```
['AP-1', 'ATF', 'C/EBP', 'Oct-1', 'c-Myc']
```

In [ ]:
```
#order
sorted(TF_motif.values())
```

Out[ ]:
```
['atgcaaat', 'attgcgcaat', 'cacgtg', 'tga(g/c)tca', 'tgacgtca']
```

7. MODULES AND PACKAGES

Modules in Python are simply Python Files with the .py extension, which contain deFinitions of functions, or variables, usually related to a speciFic theme.

Packages group multiple modules under one name, by using “dotted module names”. For example, the module name A.B designates a submodule named B
in a package named A

In [ ]:
```
#use    the     sys.path        variable        from    the     sys     built-in mod
import sys
sys.path
```

In [ ]:
```
sys.path.append("C:\\Users\\Azka") #you can     extend  it by
#sys.path
```

In [ ]:
```
import dnautil
```

```
dna="atgaggcggcggcggccgcggctaggt"
dnautil.gc(dna)
```

Package Example

You can even have other packages inside your package:

bioseq/

```
    __init__.py
    dnautil.py
    rnautil.py
    proteinutil.py
    fasta/
        __init__.py
        fastautil.py
    fastq/
        __init__.py
        fastqutil.py
```

To use the module dnautil "bioseq.dnautil.gc(dna)"

8. IFS AND LOOPS

In [ ]:
```
dna=input('Enter DNA sequence:')
if 'n' in dna :
        nbases=dna.count('n')
        print("dna sequence has %d undefined bases " % nbases)
```

Comparison Operators

> 'a'=='A'

False

> 'GT' != 'AG'

True

> 'A'<'C'

True

> 10+1==11

True

Membership Operators

> in | not in

Identity Operators

> is | not is

In [ ]:
```
alphabet=['a','b','g','t','e']

newalphabet=alphabet[:]
```

In [ ]:
```
alphabet == newalphabet
```

Out[ ]:
```
True
```

In [ ]:
```
alphabet is newalphabet
```

Out[ ]:
```
False
```

Alternative Execution

In [ ]:
```
dna=input('Enter DNA sequence:')

if 'n' in dna :
        nbases=dna.count('n')
        print("dna sequence has %d undefined bases " % nbases)
else:
        print("dna sequence has no undefined bases")
```

```
dna sequence has no undefined bases
```

Multiple Alternative Executions

In [ ]:
```
dna=input('Enter DNA sequence:')

if 'n' in dna :
        print("dna sequence has undefined bases ")
elif "N" in dna :
        print("dna sequence has undefined bases ")
else:
        print("dna sequence has no undefined bases")
```

```
dna sequence has no undefined bases
```

Logical Operators

> and - True if both conditions are true
>
> or - True if at least one condition is true
>
> not - True if condition is false

In [ ]:
```
dna=input('Enter DNA sequence:')

if 'n' in dna or "N" in dna:
        nbases=dna.count('n')+dna.count("N")
        print("dna sequence has %d undefined bases " % nbases)
else:
        print("dna sequence has no undefined bases")
```

```
dna sequence has no undefined bases
```

The while Loop

In [ ]:
```

```

```
dna=input('Enter DNA sequence:')
pos=dna.find('gt',0) # position of donor splice site
while pos>-1 :
        print("Donor splice site candidate at position %d" %pos)
pos=dna.find('gt',pos+1)
```

The for Loop

In [ ]:
```
motifs=["attccgt","agggggtttttcg","gtagc"]
for m in motifs:
        print(m,len(m))
```

```
attccgt 7
agggggtttttcg 13
gtagc 5
```

The range() Function

The range() built-in function allows you to iterate over a sequence of numbers

In [ ]:
```
for i in range(4):
        print(i)
```

```
0
1
2
3
```

In [ ]:
```
for i in range(1,10,2):
        print(i)
```

```
1
3
5
7
9
```

Problem. Find if all characters in a given protein sequence are valid amino acids.

In [ ]:
```
protein='SDVIHRYKUUPAKSHGWYVCJRSRFTWMVWWRFRSCRA'
for i in range(len(protein)):
        if protein[i] not in 'ABCDEFGHIKLMNPQRSTVWXYZ':
            print("protein contains invalid amino acid %s at position %d"%(protein[i
```

```
protein contains invalid amino acid U at position 8
protein contains invalid amino acid U at position 9
protein contains invalid amino acid J at position 20
```

Problem. Suppose we are only interested in Hinding if a protein sequence is valid, not where are all the invalid characters in the sequence.

In [ ]:
```
protein='SDVIHRYKUUPAKSHGWYVCJRSRFTWMVWWRFRSCRA'
for i in range(len(protein)):
        if protein[i] not in 'ABCDEFGHIKLMNPQRSTVWXYZ':
            print("this is not a valid protein sequence!")
            break
```

```
this is not a valid protein sequence!
```

The continue Statement

The continue statement causes the program to continue with
the next iteration of the nearest enclosing loop, skipping the rest of the code in the loop.

Problem. Delete all invalid amino acid characters from a protein sequence.

In [ ]:
```
protein='SDVIHRYKUUPAKSHGWYVCJRSRFTWMVWWRFRSCRA'
corrected_protein=''
for i in range(len(protein)):
        if protein[i] not in 'ABCDEFGHIKLMNPQRSTVWXYZ':
            continue
        corrected_protein=corrected_protein+protein[i]

print("Corrected protein sequence is:%s"%corrected_protein)
```

```
Corrected protein sequence is:SDVIHRYKPAKSHGWYVCRSRFTWMVWWRFRSCRA
```

In [ ]:
```
#An     Example Of      Using   else    With    A       for     Loop
#Find   all     prime   numbers smaller than    a       given   integer.
N=10
for y in range(2, N):
    for x in range(2, y):
        if y % x == 0:
            print(y, 'equals', x, '*', y//x)
            break
    else:
        # loop fell through without finding a factor
        print(y, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

The pass Statement

Python's pass statement is a placeholder. it does nothing.
It is used when a statement is required syntactically but you do not want any command or code to execute.

In [ ]:
```
if motif not in dna:
    pass
else:
    print(motif,dna)
```

9. FUNCTIONS

A function is a part of a program. It takes a list of argument
values, performs a computation with those values, and returns a single result. like print(), len() etc.

Some Useful DNA Sequence Functions

1. A function that computes the GC percentage of a DNA sequence
2. A function to check if a DNA sequence has an in frame stop codon.
3. A function to reverse complement a DNA sequence.

In [ ]:
```
#A      Function        To      Compute The     GC      Percentage      Of      A
def gc(dna) :
    "this function computes the GC percentage of a dna sequence"
    nbases=dna.count('n')+dna.count('N')
    gcpercent=float(dna.count('c')+dna.count('C')+dna.count('g')
+dna.count('G'))*100.0/(len(dna)-nbases)
    return gcpercent
gc('AAAGTNNAGTCC')
```

Out[ ]:
```
40.0
```

In [ ]:
```
#execution is easy
gc(dna)
```

Out[ ]:
```
47.05882352941177
```

In [ ]:
```
# function that finds if it has a stop codon in a frame
def has_stop_codon(dna,frame) :
        'This function checks if given dna sequence has in frame stop codons.'
        stop_codon_found=False
        stop_codons=['tga','tag','taa']
        for i in range(frame,len(dna),3) :
            codon=dna[i:i+3].lower()
            if codon in stop_codons :
                stop_codon_found=True
            break
        return stop_codon_found
```

In [ ]:
```
dna=input("Enter a DNA sequence, please:")
if has_stop_codon(dna,frame=0):
    print("Input sequence has an in frame stop codon.")
else :
    print("Input sequence has no in frame stop codons.")
```

```
Input sequence has no in frame stop codons.
```

In [ ]:
```
dna="aaatgagcggccggct"
has_stop_codon(dna,frame=1)
has_stop_codon(dna,1)
```

Out[ ]:
```
False
```

In [ ]:
```
seq='tgggcctaggtaac'
has_stop_codon(seq,1)
```

Out[ ]:
```
False
```

In [ ]:
```
has_stop_codon(frame=0,dna=seq)
```

Out[ ]: False

In [ ]:
```
has_stop_codon(seq,frame=2)
```

Out[ ]: False

In [ ]:
```
# defining function for reverse complementing a base

def reverse_string(seq):
    return seq[::-1]
def complement(dna):
    """Return the complementary sequence string."""
    basecomplement = {'A':'T', 'C':'G','G':'C', 'T':'A', 'N':'N',
     'a':'t', 'c':'g','g':'c', 't':'a', 'n':'n'}
    letters = list(dna)
    letters = [basecomplement[base] for base in letters]
    return ''.join(letters)
def reversecomplement(seq):
        """Return the reverse complement of the dna string."""
        seq = reverse_string(seq)
        seq = complement(seq)
        return seq
```

In [ ]:
```
reversecomplement('CCGGAAGAGCTTACTTAG')
```

Out[ ]: 'CTAAGTAAGCTCTTCCGG'

List Comprehensions

List comprehensions in Python provide a concise way to create lists.

Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence, or to create a subsequence of those elements that satisfy a certain condition.

In [ ]:
```
# How   To      Complement(only)        Each    Letter  In      A       DNA     Sequ
dna='AGTGTGGGGCG'
basecomplement = {'A':'T', 'C':'G','G':'C', 'T':'A', 'N':'N',
'a':'t', 'c':'g','g':'c', 't':'a', 'n':'n'}
letters=list(dna)
letters = [basecomplement[base] for base in letters]
letters
```

Out[ ]: ['T', 'C', 'A', 'C', 'A', 'C', 'C', 'C', 'C', 'G', 'C']

Split And Join

Split and join are methods of the string object.

Split The method split() returns a list of all the words in the string.

Join The method join() returns a string in which the string elements were joined from a list. The separator string that joins the elements is the one upon which the

function is called.

In [ ]:
```
# split
sentence="enzymes and other proteins come in many shapes"
sentence.split()
```

Out[ ]:
```
['enzymes', 'and', 'other', 'proteins', 'come', 'in', 'many', 'shapes']
```

In [ ]:
```
# join
'-'.join(['enzymes', 'and', 'other', 'proteins',
'come', 'in', 'many', 'shapes'])
```

Out[ ]:
```
'enzymes-and-other-proteins-come-in-many-shapes'
```

10. WITH THE OUTSIDE

Reading and Writing Files

To read or write files use the built-in function open (filename, mode)

> f=open('myfile','r')
>
> f=open('myfile','w')
>
> f=open('myfile','a')

In [ ]:
```
#Reading        from    a       file
try:
    f = open("myfile")
except IOError: #Errors When    Opening a       File
    print("the file myfile does not exist!!")
```

In [ ]:
```
#Reading        From    a       File
for line in f:
    print(line)
```

```
This is the first line of the file.

Second line of the file.This is a third line
```

In [ ]:
```
f.seek(0)
f.read()
```

Out[ ]:
```
'This is the first line of the file.\nSecond line of the file.This is a third line'
```

In [ ]:
```
#Changing       Positions       Within  a       File    Object
f.seek(0)
f.readline()
```

Out[ ]:
```
'This is the first line of the file.\n'
```

In [ ]:
```
# Writing       Into    a       File
```

```
f = open('c:\\Users\\Azka\\Desktop\\GenomicDS practice\\myfile','a')
f.write('This is a third line')
```

In [ ]:
```
# Writing       Into    a       File
f = open("myfile")

for line in f:
    print(line)
```

```
This is the first line of the file.

Second line of the file.This is a third line
```

In [ ]:
```
# Closing       a       File    Object
f.close()
```

Reading a FASTA File

Exercise: Build a dictionary containing all sequences from a FASTA file.

In [ ]:
```
# Reading       a       FASTA   File
try:
    f = open("fastafile")
except IOError:
    print("File myfile.fa does not exist!!")
seqs={}
for line in f:
    # let's discard the newline at the end (if any)
    line=line.rstrip()
    # distinguish header from sequence
    if line[0]=='>': # or line.startswith('>')
        words=line.split()
        name=words[0][1:]
        seqs[name]=""
    else : # sequence, not header
        seqs[name] = seqs[name] + line
f.close()
```

In [ ]:
```
# Retrieving    Data    From    Dictionaries
for name,seq in seqs.items():
    print(name,seq)
```

In [ ]:
```
# Reading       the     Command Line    Arguments       in      processfasta.py

"""
processfasta.py builds a dictionary with all sequences
from a FASTA file.
"""
import sys
filename=sys.argv[1]

try:
    f = open("fastafile")
except IOError:
    print("File %s does not exist!!"%filename)
```

```
seqs={}
for line in f:
    # let's discard the newline at the end (if any)
    line=line.rstrip()
    # distinguish header from sequence
    if line[0]=='>': # or line.startswith('>')
        words=line.split()
        name=words[0][1:]
        seqs[name]=""
    else : # sequence, not header
        seqs[name] = seqs[name] + line
f.close()
```

In [ ]:

```
# # Command     Line    Arguments
# # python processfasta.py fastafile
# import sys
# print(sys.argv)
# #processfasta.py -l 250 fastafile
```

In [ ]:

```
# # Using       the     System  Environment
# sys.stdin.read()
# a line
# another line
```

In [ ]: