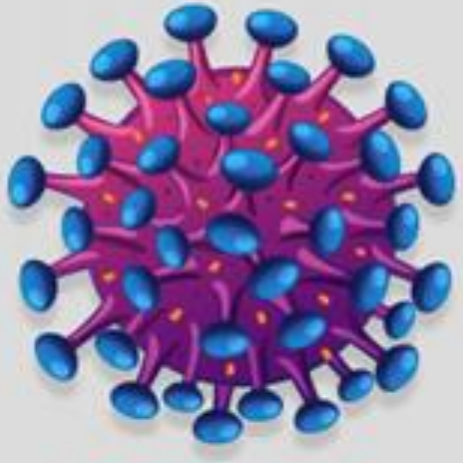
bioInformatics

The objective of this course introduce and learn students some software used for simulate and analysis microbiology data.

Dr Khodja



HIV

HUMAN
IMMUNODEFICIENCY VIRUS

HIV IS A VIRUS WHICH
ATTACKS IMMUNE SYSTEM
IN HUMANS

Simian immunodeficiency **virus** (**SIV**)

Evolutionary Tree

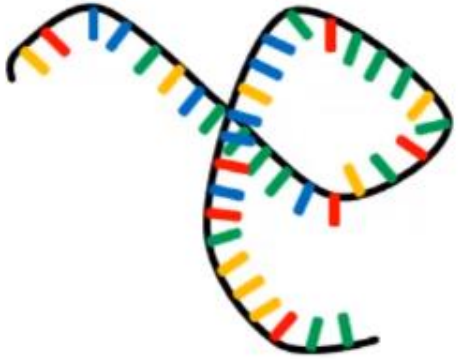
SIVs (monkeys)
HIV (human)
human infection

human HIV/M
human HIV/M
chimpanzee SIV
chimpanzee SIV
human HIV/N
human HIV/N
chimpanzee SIV
chimpanzee SIV
chimpanzee SIV
chimpanzee SIV
chimpanzee SIV
chimpanzee SIV
human HIV/O
human HIV/O
chimpanzee SIV
chimpanzee SIV
red-capped manabey SIV
drill SIV
vervet monkey SIV
tantalus monkey SIV
sooty mangabey SIV
human HIV/A
human HIV/B
sooty mangabey SIV
Sykes's monkey SIV
greater spot-nosed monkey SIV

GENE-PROTEIN FLOW

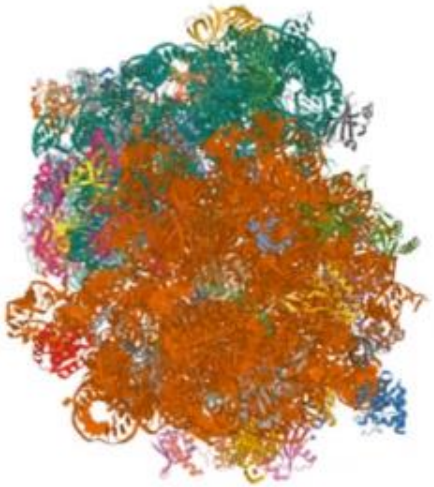
DNA

Transcription



RNA

Translation



PROTEIN

Splicing

1.6 Structural bioinformatics

"Structural bioinformatics is a subset of Bioinformatics that deals with the prediction and analysis of 3D structure of Macromolecules such as DNA, RNA, and Proteins." And the second thing which comes is why understanding the structure of macromolecule is important. The reasons are first: structure determines function, so learning structure helps in understanding of function. Secondly the Structure is more conserved than the sequence, hence enabling identification of a much more distant evolutionary relationship. Thirdly understanding the structural determinants enables the design and modification of proteins for industrial and medical benefit. The structural bioinformatics field and concepts related to it offers not only a way of coordinating views about sequence-structure-function questions but also a mechanism for detecting unobserved behavior and proposing novel experiments (Konings et al., 1987; Schuster et al., 1994).

Programming Strategies*

1. Identify the required inputs, such as data or specifications from the user.
2. Make an overall design for the program, including listing all the steps by which the program computes the output.
3. Decide what will be the output of the program: will the result be in a file, or displayed on the screen?
4. Refine the overall design by specifying more detail.
5. Write the program.

* Adapted from *Beginning Perl for Bioinformatics* by James Tisdall, Published by O'Reilly Media, Inc., 2001

Designing a Program

Write *pseudocode* for a program that computes the GC percentage composition of a DNA sequence:

- *read DNA sequence from user*
- *count the number of C's in DNA sequence*
- *count the number of G's in DNA sequence*
- *determine the length of the DNA sequence*
- *compute the GC%*
- *print GC%*

A First Program In Python

```
read DNA sequence from user
>>> dna = 'acgctcgcgcggcgatagctgatcgatcggcgcgctttttttttaaaag'
count the number of C's in DNA sequence
>>> no_c=dna.count('c')
count the number of G's in DNA sequence
>>> no_g=dna.count('g')
determine the length of the DNA sequence
>>> dna_length=len(dna)
compute the GC%
>>> gc_percent=(no_c+no_g)*100.0/dna_length
print GC%
>>> print(gc_percent)
53.06122448979592
```

The .0 after 100 is only required in Python 2.x

Adding Comments

gc.py

```
#! /usr/bin/python
"""                    ← multiple line comments are included in between """ ..."""
This is my first Python program.
It computes the GC content of a DNA sequence.
"""
# get DNA sequence:    ← everything that follows a # is ignored up to the line end
dna = 'acgctcgcgcggcgatagctgatcgatcggcgcgctttttttttaaaag'
no_c=dna.count('c') # count C's in DNA sequence
no_g=dna.count('g') # count G's in DNA sequence
dna_len=len(dna) # get the length of the DNA sequence
gc_perc=(no_c+no_g)*100.0/dna_len # compute gc percentage
print(gc_perc) # print GC% to screen
```

Datatypes and operators

Datatypes

- int (integers or whole numbers)
- float (decimal numbers or floating-point numbers)
- bool (Boolean or True/False)
- str (string or a collection of characters like a text)
- list, Tuple, set (Collection of Items

Table Assigning variables in Python.

Code	Output
`weight = 75` `print(weight)`	`75`

Table : Operations with variables.

Code	Output

`weight = 75`
`height = 1.5`
`bmi = weight/height`
`print(bmi)`
	`50.0`

Operations on strings

There are a few ways to concatenate or join strings. The easiest and most common way to add join strings is to use the plus symbol (+). i.e., in simplest terms, merely adding them.

The "+" operator can be used to combine any number of strings. A critical point to remember is that when adding strings, all datatypes must be strings; for example, if users add a string with an integer, such as "PLANT"+4, an error message indicates that the "str" type and the "int" type cannot be added. To add a number, it must first convert it to the "str" type using the str (number) function. While integers and strings cannot be added, the same string can be printed several times using the "*" operator and a "int" datatype. For instance, "PLANT"*2 returns the string twice, i.e., "PLANTPLANT".

Table Few methods in strings.

Code	Output

`peptide = 'TSLWGLLFLSAALSLWPTSG'`
`print(peptide.count('A'))`
`print(peptide.find('LW'))`
`print(len(peptide))`
	`2`
	`2`
	`20`

Using Python As A Calculator

```
>>> 5+5
10
>>> 10.5-2*3
4.5
>>> 10**2
100
>>> 17.0 // 3
5.0
>>> 17 % 3
2
>>> 5 * 3 + 2
17
```

** is used to calculate powers

floor division discards the fractional part

the % operator returns the remainder after division

* takes precedence over +, -

Accessing values in list

Like strings, list items also have indexes starting with "0" for forward indexing and "−1" for backward indexing

The items inside a list can be accessed using brackets [] and indexes.

Slicing a list allows to access a subset of it. The string slice operator can also slice lists. Similarly, to string, omitting the first index causes the slice to begin at the beginning. If the second is absent, the slice ends. If both of them are removed, the slice is a copy of the List

The "+" operator can be used to concatenate two lists and the "*" operator to repeat a list any specified number of times.

	['Moss'	'Embryophyte'	'Thallophyte'	'Conifer']
Forward indexing	**0**	**1**	**2**	**3**
Backward Indexing	**-4**	**-3**	**-2**	**-1**

Table List slicing.

Code	Output

`# not including index 2`
`print(plants[0:2])`
`# everything up to index 3`
`print(plants[:3])`
`# index 1 to end of list`
`print(plants[1:])`
`# Coping whole list`
`print(plants[:])`
	`['Moss', 'Embryophyte']`
	
	`['Moss', 'Embryophyte', 'Thallophyte']`
	
	`['Embryophyte', 'Thallophyte', 'Conifer']`
	
	`['Moss', 'Embryophyte', 'Thallophyte', 'Conifer']`

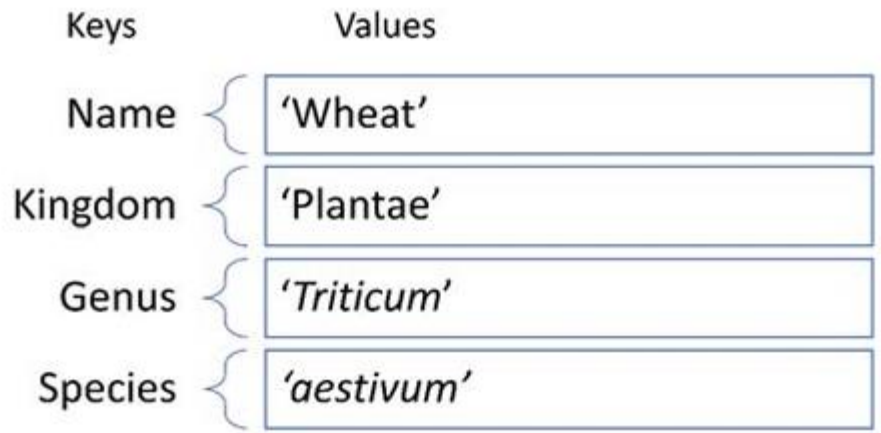
Methods with lists

Python provides some in-built methods for List such as:

- count () methods will return the total number of occurrences of an item in the List.
- index () will give the index of an item.
- append () adds an item at the end of the List.
- remove () will remove the first occurrence of the item in the List.
- pop () will remove the item at index provided by the user.
- min (), max () and sum () will provide the minimum, maximum and sum of the lists constituting number values.
- len () will provide the total number of items in the List.
- sort () method can be used to sort a list of numerical values in increasing or decreasing order, or a list of string in A-Z or Z-A order.

Dictionary in Python

Dictionaries are data structures in Python that are similar to hash tables or hashmaps in other computer languages. Each key corresponds to a single value in a dictionary. The ideal approach to establish a dictionary is to put the key:value pairs inside curly brackets "{}". Only "{}" can declare an empty dictionary



Python dictionary key: Value pairs.

Table Creating a Python dictionary.

Code	Output

`crop = {}`
`crop = {'Name':'Wheat','Kingdom':'Plantae','Genus':'Triticum','Species':'aestivum'}`
`print(crop)`
`print(type(crop))`
	`{'Name': 'Wheat', 'Kingdom': 'Plantae', 'Genus': 'Triticum', 'Species': 'aestivum'}`
	`<class 'dict'>`

Conditional statements

Until now, the programmes are simple, not clever, and not making decisions. Conditional statements are required to make a program make decisions based on conditions. Computers have only two states, True or False, like a light switch has two states, On or Off. In Python, these True/False situations are known as booleans.

A condition is always defined by comparison, such as larger than, less than, or equal. Here are some comparisons with Python operators:

- Equal: a == b
- Not Equal: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

All these comparisons result in Boolean values "True" or "False".

Logical operators

When comparing many conditions, logical operator are used. The logical operators "and", "or", and "not" are the same in Python as in English. Logic operators usually work on conditions.

"and" operator will only give true if both the conditions are true, "or" will give true if either of the conditions is true, lastly "not" will give just the opposite condition, i.e., it will give false for true and true for false.

Table If statements.

Code	Output
control_expression = 14 treated_expression = 3.5 if control_expression > treated_expression: print(‘downregulated’)	downregulated

Table If, else statement.

Code	Output

control_expression = 14
treated_expression = 3.5
if control_expression > treated_expression:
    print('Gene is downregulated')
else:
    print('Gene is upregulated')
	Gene is downregulated

Table Simple while loop.

Code	Output

`a = 0`
`while a<6:`
`  print(a)`
`  a = a+1`
	0
	1
	2
	3
	4
	5

Table Simple “for” loop.

Code	Output

`plants = ['Moss', 'Embryophyte', 'Thallophyte','Conifer']`
`for plant in plants:`
`  print(plant)`
	Moss
	Embryophyte
	Thallophyte
	Conifer

Table Python code snippets showing the breaking a loop before its ending.

Code	Output

`for temp in plants:`
`  print(temp)`
`  if temp == 'Thallophyte':`
`  break`
	Moss
	Embryophyte
	Thallophyte

Copy of basics1.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

type using print (type ('khodja '))

```
print( type (5))
print(type   (3.5))
print  (type  (3 + 2j))
print  ( type  ("khodja "))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
<class 'str'>
```

```
dna="gatccccccgatattatttgc"
dna1= dna
dna2=dna1.upper()
print(dna2)
```

```
GATCCCCCCGATATTATTTGC
```

compute the gc % and show it in the monitor

```
seq='acgctcgcgcggcgatagctttgggttt'
cn=seq.count ('c')
gn =seq.count  ('g')
l=len(seq)
gcper=100*(cn +gn )/l
print ('the percentage of gc equals ',gcper)
```

```
the percentage of gc equals  60.714285714285715
```

Copy of basics1.ipynb

File Edit View Insert Runtime Tools Help Last e

+ Code + Text

- copy string (replicate) in membership not in

```
'atc'+'gcg'
```

```
'atcgcg'
```

```
"atg"*3
```

```
'atgatgatg'
```

```
'atc'*3
```

```
'atcatcatc'
```

```
'atg' in 'atggccggcgta'
```

```
True
```

```
'n' in 'atgtgggg'
```

```
False
```

Copy of basics1.ipynb

File Edit View Insert Runtime

+ Code + Text

```
dna="gatccccccgatattatttgc
dna[0]
dna[-1]
dna[-2]
dna[0:3]
print (dna[:3])
dna[2:]
len(dna)
dna.count('c')
dna.count ('gc')
dna1= dna
dna2=dna1.upper()
print (dna)
print (dna1 )
print  (dna2 )
dna.find ('gc')
dna.find('tat')
dna.find ('gat',15)
dna.islower ()
dna.isupper()
dna.replace ('c','G')
```

```
gat
gatccccccgatattatttgc
gatccccccgatattatttgc
GATCCCCCCGATATTATTTGC
```