INDICE

PRIMA PARTE: PROGRAMMAZIONE PYTHON

- Python
  - Python Structural Overview
  - Data Types: Basic and Composite
  - Assignment Operation
  - Control Flow: Conditionals and Iteration
  - Functions
  - Exceptions
  - File I/O
  - Bits of Functional-Style Programming
  - Modules
  - Blocks
  - Random Numbers Generators
- Object-Oriented Python
- Numpy and Matplotlib
- Regular Expressions
- Biopython
- Albero binario

SECONDA PARTE: TEORIA

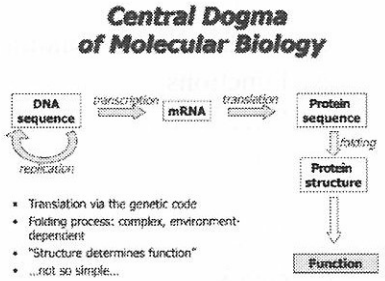
- Handling of Nucleic Acid Sequences
- Algorithm and Computational Complexity
- Working with Strings:
  - Basic Motif Search
  - Pairwise Sequences Allignements
  - From Molecular Evolution to Scoring Models
  - Sequence Alignment in Pratice
- Biological Data Banks
- Assembly and Shortest Common Superstring
- Basics on Protein Structure
- Molecular Mechanics and Force Fields
- Simulation and Simulators
- Protein Models

master COPY

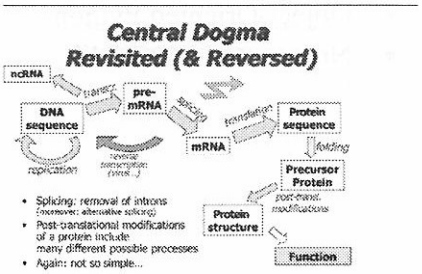
Bioinformatica: settore multidisciplinare (biologia e informatica) in cui bisogna avere competenze di entrambe.
L'approccio dei biologi ad un problema prevede una raccolta di campioni e una successiva classificazione; non guardano al problema in sé ma analizzano i risultati. Y=f(x)
L'approccio informatico è per modelli, quindi si guarda alla funzione.
In sintesi, in bioinformatica si affrontano i problemi biologici con approccio informatico.
Bioinformatica a diverse scale:

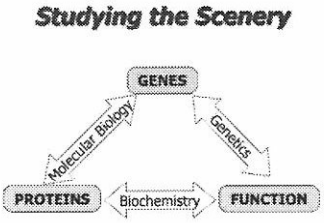
Corpo umano:	Chirurgia-Robotica
Tessuti-organi:	Simulazione biomeccanica
Cellule:	Sistemi in grado di simulare processi/interazione cellulari.
Molecole:	Studio delle strutture molecolari, movimenti e interazioni.
Geni:	Analisi delle sequenze

(molecole e geni sono quelle di interesse nel corso)
Dogma centrale della biologia molecolare:
Flussi di informazione all'interno di un organismo biologico.
Le informazioni per riuscire a regolare l'organismo sono contenute nel DNA, ma per essere utilizzate bisogna costruire, dalla codifica, delle molecole che trasportano l'informazione (mRNA). Ciò si fa tramite la trascrizione.
Dall'mRNA, tramite sintesi, si crea una sequenza proteica che, ripiegandosi, crea una struttura proteica, depositaria della funzionalità di quel soggetto.
Il folding è un processo complicato; non si può predire in modo semplice e immediato, a partire dalla sequenza.



In realtà questo flusso è più complesso: tipicamente (negli eucarioti) la trascrizione da DNA a mRNA non è diretta, ma vi è un pre-mRNA, e c'è una fase di splicing, dove alcune parti vengono rimosse (perché ci sono zone più informative e altre meno).
ncRNA: non coding RNA; hanno funzionalità diversa, ad esempio enzimatica, ecc.





Per studiare l'intero panorama vi sono diverse branche.
Biologia molecolare: geni-proteine
Biochimica: proteine-funzione
Genetica: geni-funzione

Gli organismi evolvono perché le molecole alla loro base possono cambiare in base a tanti fattori. Ci possono essere sequenze che, pur se diverse, sono legate da meccanismi di evoluzione molecolare.
Se il ripiegamento delle proteine non avvenisse in modo corretto, bisognerebbe capirne la causa e le conseguenze di ciò.
Il linguaggio informatico che si utilizzerà è Python: flessibile, lavora bene con le sequenze ed è tra i più utilizzati.

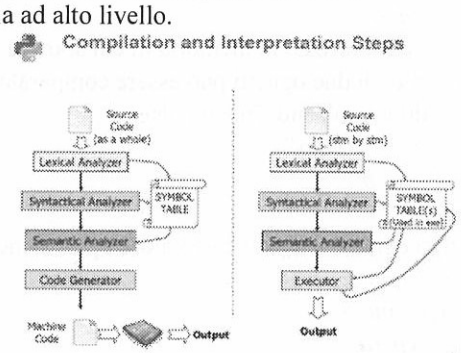
Programmazione Python
-codice leggibile: sintassi naturale e 'blocchi di rientro ' che forzano la corretta strutturazione e leggibilità del codice;
-riutilizzo codice: molte librerie presenti;
-programmazione object oriented: gestione di programmi molto complessi;
-prestazioni elevate: NumPy (libreria numerica) provvede a creare matrici algebriche molto veloci, e può scaricare alla velocità di C i moduli ad alta intensità. Ciò è vero all'interno della sua categoria di linguaggio (strutturalmente non è così efficiente come ad esempio il linguaggio C).

Differenza tra compilazione e interpretazione:

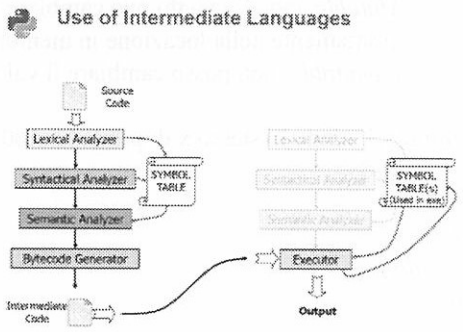
- La compilazione consiste nell'analizzare il programma che ho scritto e ottenerne un equivalente con istruzioni accettabili dalla CPU (trasformazione da dati ad alto livello ad operazione per la CPU).

- L'interpretazione identifica le singole istruzioni del programma e si fanno eseguire dalla CPU un passo alla volta. Ho bisogno di un programma che vede qual è l'istruzione successiva da dare in pasto alla CPU.

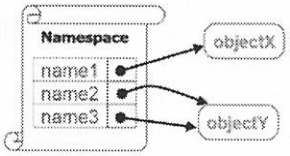
In entrambi gli approcci si parte da un codice sorgente, ovvero il programma ad alto livello.
Nella compilazione, si passa dal codice sorgente ad un'analisi lessicale e una sintattica e si depositano le informazioni in una struttura dati nota come tabella dei simboli. Si ha poi l'analisi semantica seguita dalla generazione del codice da dare alla CPU e che permette di ottenere l'output.



Nell'interpretazione i passi sono molto simili, infatti, anche qui si raccolgono le informazioni nella tabella dei simboli (analisi lessicale e sintattica); si ha l'analisi semantica e poi si ha l'esecutore che si riferisce alla tabella dei simboli per sapere quale sarà l'istruzione successiva, oltre che tener traccia del programma. Si ottiene dunque l'output.
La compilazione risulta particolarmente efficiente ma non molto flessibile.
L'interpretazione invece ha velocità minore, quindi basse prestazioni perché lavora passo-passo, però ha massima flessibilità.
Io vorrei un sistema contemporaneamente efficiente e flessibile; si parla di linguaggi intermedi. L'idea è quella di fare compilazione per aver una versione del programma vicino al codice macchina più l'utilizzo di un interprete che prenderà in ingresso il codice intermedio. L'esecutore ha sempre bisogno di salvare i dati per sapere quali saranno le istruzioni da eseguire successivamente.
La flessibilità sta nell'utilizzare l'executor per macchine diverse: portabilità del codice (il codice intermedio è sempre lo stesso, mentre l'esecutore varia per ogni sistema).



Linguaggio di scripting: è un linguaggio utilizzato per controllare le applicazioni (il linguaggio di programmazione invece è finalizzato allo sviluppo di programmi).
Il sistema operativo è chiamato shell script.
Gli script (linguaggi) sono spesso interpretati dal codice sorgente o compilati dal bytecode, la rappresentazione intermedia.
Caratteristiche di Python: è un linguaggio semplice, minimale, che permette di concentrarsi sulla soluzione del problema. È facile da imparare, portabile (i programmi possono lavorare su piattaforme senza cambiarli del tutto), multiparadigma (scomporre la complessità del problema in problemi più piccoli e più semplici), object oriented, estensibile, incorporabile (in altri programmi C/ C++), e ricco di librerie.
Interpretazione in Python: l'interprete esegue le istruzioni scritte in uno script; si ha in input un file con le istruzioni le quali vengono analizzate una alla volta, si ottiene il bytecode che viene immediatamente eseguito.
Come faccio partire l'interprete Python? Vado nella shell del sistema operativo, scrivo Python e invio;
>>> pronto per i comandi
Ctrl+D rimuovo l'interprete Python.
Se scrivo 7/3 l'interprete interpreta l'operazione come una divisione tra numeri interi e da come risultato 2.
I commenti sono preceduti dal carattere '#'.
Se volessi eseguire un programma già scritto faccio:
python 'nome programma' e invio.
Spyder è l'ambiente di sviluppo che utilizzeremo (accessibile tramite Anaconda), versione 2.7 .



Panoramica strutturale di Python
Qualunque entità gestita (handled entity) è un oggetto, o un nome o un namespace.
L'oggetto è destinato a contenere dati e/o funzionalità.
I nomi sono le entità utilizzate per riferire gli oggetti; un oggetto può essere riferito da più nomi.
Se l'oggetto non è riferito allora non è utilizzabile e viene direttamente collocato nel garbage-collector (come in Java, è il luogo in cui si 'buttano' le cose inutilizzate).
Il namespace è l'entità in cui si collezionano i nomi.
L'identità è l'indirizzo dell'oggetto in memoria, e si definisce al momento della creazione.

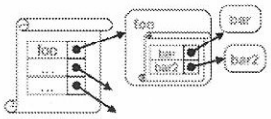
L'identità di un oggetto *x*, si ottiene invocando *id(x)*.
>>> x=100
>>> id(x)
108 #è l'indirizzo in memoria in cui si trova x
l'identità di due oggetti può essere comparata tramite *is,* cioè mi dice se i nomi riferiscono lo stesso oggetto, restituisce un valore booleano True o False:
>>> a=b= "ciao"
>>> a is b
True
Il tipo definisce il possibile valore/operazione per l'oggetto:
>>> type(x)
< type 'int'>
>>>type(a)
<type 'str'>
Il valore di un oggetto può essere:

- *Mutable*, cioè l'oggetto può cambiare valore senza cambiare la sua identità, cioè il cambiamento si fa direttamente nella locazione in memoria;
- *Immutable*, non posso cambiare il valore dell'oggetto in memoria.

Esempio: riprendo lo stesso x di prima, x=100
>>> x=x+1
>>>x
101
>>> id(x)
109

In genere un oggetto è riferito da un nome, se non viene più riferito viene automaticamente eliminato tramite garbage collector.
Un oggetto può riferire altri oggetti, chiamati attributi, e ogni oggetto ha un namespace associato per i suoi attributi.

Esempio: L'oggetto foo ha come attributi bar e bar2. A livello strutturale si ha un namespace che contiene l'oggetto foo, e poi il namespace associato di foo che contiene i due elementi bar e bar2.



foo.bar per dire che bar sta dentro foo; è il modo di ricercare gli attributi.
Gli oggetti possono contenere dati o funzionalità. Se corrisponde ad una funzione, l'attributo viene chiamato metodo -> funzione associata ad un oggetto.
All'interno della funzione si usano delle variabili locali, quindi esistono namespaces temporanei per supportare le variabili al momento dell'invocazione della funzione.
Regole per determinare in quale namespace si trovano i nomi:

- Le definizioni all'interno di funzioni e metodi si trovano all'interno del namespace locale.
- Le definizioni nei moduli finiscono nel namespace dell'oggetto modulo.
- Le definizioni in una classe sono nel namespace degli attributi della classe.
- In un'istanza di classe, quando un attributo non si trova nel namespace dell'oggetto, si ricerca nel namespace della classe. Questo è il modo in cui normalmente si cercano i metodi.

Oggetti di tipo *Callable*: possono essere richiamati; supportano la chiamata di funzione effettuata con le parentesi tonde (). In questa categoria di oggetti rientrano le funzioni e oggetti come liste, tuple, interi.

Esempio: mi crea una lista in memoria; il sistema prende una stringa e crea la lista con i singoli caratteri della stringa che ho messo tra parentesi:
>>> a=list('ciao')
>>>a
['c', 'i', 'a', 'o']
Altri oggetti Callable: oggetti di tipo class, che servono per specificare tipi custom; supportano la chiamata di funzione per creare istanze. Infine, anche i metodi sono oggetti callable.

Tipi di dati

Ci sono due grandi categorie: basici e composti (che aggregano più valori).

Basici: interi, floating-point (virgola mobile), interi lunghi (servono più byte per rappresentarli), numeri complessi, stringhe, tipo e funzioni.

Un valore speciale è "None", che serve per non riferire nulla, cioè nessun valore; se voglio che una variabile non riferisca niente, si crea in memoria None e si fa in modo che la variabile riferisca None. Per vedere se riferisce a None si fa: *x is None* oppure *x == None* (N va scritta in maiuscolo, key sensitive, e lo stesso vale anche per True e False, T e F iniziali sono maiuscole).

Composti: sono chiamati anche strutture dati di tipo contenitivo, perché al loro interno possono contenere altri oggetti.

Tra i tipi contenitore abbiamo:

- Lista, sequenza di elementi, che può essere modificata;
- Tupla, lista speciale, che non viene modificata;
- Dizionari, per immagazzinare informazioni tramite una keyword, che identifica quelle info (Dicts, Maps, Associative Array);
- Set (insiemi), in cui non è definito un ordinamento degli elementi;

La funzione len(x) restituisce il numero di elementi del contenitore.

Non è prevista la struttura dati per gli array, cioè non fanno parte del nucleo del linguaggio.

Valori da specificare per tali tipi di dati: un letterale è una sequenza di caratteri del linguaggio che corrisponde ad un valore.



Esempi:

n = 25 n riferisce il valore 25 di tipo intero

n = 034 notazione ottale (0 davanti al numero intero)

n = 0x4f notazione esadecimale

longn = 135454L

Una variabile non ha un tipo esplicito associato; il programma, infatti, ispeziona il letterale e decide il tipo dell'oggetto da costruire.

Per i floating-point i letterali sono specificati con un punto:

f1 = 4.0

f2 = 4.2E-12 $(4.2*10^{-12})$

Letterali complessi:

cn = 10+4j Re{10} Im{4}

cn = 10 + 0j anche se Im è nulla, scrivo 0j per considerarlo comunque un numero complesso

Stringa (sequenza di caratteri):

I delimitatori di Python possono essere diversi cioè ' ' (apici singoli) o " " (apici doppi), oppure sequenze di 3 apici singoli o 3 apici doppi.

Citazione singola: *s = 'spam'* oppure *s = "spam"*

Citazione tripla: *s1= """foo"""* oppure *s2= ''' bar '''*

Con il triple quoted (citazione tripla) posso utilizzare i caratteri di delimitazione come caratteri di stringa.



Esempio:

```
>>> "What's your name?"
"What's your name?"
```

'\' può essere usato per spezzare le linee di comando.

```
>>> 'What\'s your name?'
"What's your name?"
```

Per andare a capo si scrive \n.

```
>>> print 'What\'s \nyour name?'
What's
your name?
```

Stringhe grezze: mettendo 'r' davanti alla stringa non viene processato ciò che c'è dentro la stringa.

```
>>> print r"What's your \name?"
What's your \name?
```