Programmare in Nim

Manuale da Battaglia

Ver 0.8.2

Andrea Martin

Prefazione.(V0.8.2)

Perchè Questo Manuale?

Nel momento in cui scrivo questo manuale, per Nim non ce molto materiale disponibile figurarsi poi in italiano, ne tanto meno dei manuali da tenere sempre sotto mano quando non ci si ricorda una procedura, un modulo. Certo su internet si trova tutto, ma non sempre è la cosa più pratica e veloce. Quindi ho deciso di scrivere questo manuale che non vorrà essere uno strumento molto approfondito, non ne avrei nemmeno le facoltà per farlo, ma vuol essere un qualcosa di introduttivo al linguaggio che ti permetta di capire le basi e dare un assaggio di cosa può fare Nim nel suo modo elegante ma potente; insomma un manuale da battaglia. Non mi dilungherò molto in noiose spiegazioni tecniche, ma cercherò ti fare quanto prefissato con degli esempi che spero possano pure essere divertenti, si perchè programmare in Nim è divertente!

Cosa Ti Occorre per Cominciare?

Per iniziare a creare i tuoi programmi in Nim, basta davvero poco:

- Il compilatore Nim.
- Il compilatore C tipo GCC.
- Un editor qualunque.

Il compilatore Nim puoi trovarlo al seguente indirizzo scaricabile gratuitamente: https://nim-lang.org/install.html si trovano le versioni per tutti i sistemi operativi (se usi Linux prova nel tuo gestore pacchetti, ma qui probabilmente trovi la versione più aggiornata). Ricordati poi se il sistema non lo fa automaticamente di inserire Nim nella path di sistema (per Linux ecco un esempio in .bashrc → export PATH=/home/tuoUtente/bin/nim/bin:$PATH e export PATH=/home/tuoUtente/.nimble/bin:$PATH). Ora devi installare il compilatore GCC che sicuramente è nei repositories del tuo sistema oppure MINGW per Windows che lo trovi sempre nella sezione download di Nim. Ultima cosa ti serve un editor, come spesso si consiglia quando si inizia e non si fanno cose complicate, è meglio usare un editor anzichè un ide perché diciamo ti aiuta a scrivere e fissare meglio le parole chiave e a prendere reale confidenza con il linguaggio; quindi puoi usare Kate, Notepad, Vim o quello che più ti piace. Quando hai queste cose sei praticamente apposto. In alternativa per provare il linguaggio lo si può fare comodamente in rete senza dover installare nulla (https://play.nim-lang.org/).

Parte A

Le Basi del Linguaggio.

1 - Caratteristiche Generali di Nim.

Diamo ora un rapido sguardo alle caratteristiche di Nim, una che potrebbe sembrare banale, ma non lo è l'estrema pulizia ed eleganza del codice questo oltre che a renderti piacevole la programmazione, verrà utile in seguito quando dovrai mantenere o correggere i vari errori evitandoti cosi fastidiosi mal di testa. Un'altra ottima caratteristica è che Nim segue la filosofia di Python, oltre che nello stile del codice anche in quello che potremmo definire "batterie incluse", ovvero ti fornisce già un gran numero di librerie che torneranno molto comode ed utili nelle lunghe notti davanti al pc. La compilazione di Nim è in realtà una doppia compilazione: ovvero il codice scritto in Nim viene prima convertito in C, poi da C in binario, potrebbe apparire strana la cosa, ma se ci pensiamo bene potrebbe dare molti vantaggi, il primo è che Nim genera codice ottimizzato in C poi il compilatore C (che ormai son super ottimizzati e testati) generano dei binari di ottima qualità ed efficienza, in più (ma questo è un argomento avanzato che qui non tratteremo) è estremamente facile creare dei wrapper da C. Altra caratteristica degna di nota è che Nim non ha bisogno della gestione manuale della memoria, perché lo farà per te il Garbage Collector (GC) sollevandoti cosi da un bel problema, in più Nim offre diversi sistemi di gestione, GC classico, ARC ed il nuovissimo ORC (che dovrebbe diventare di default in Nim 2.0), oppure lo si può totalmente disabilitare ma se non strettamente necessario viene sconsigliato. In Nim ce anche la metaprogrammazione (generics, template, macro) che qui vedrai solo in parte in quanto argomento avanzato. Nim usa la tipizzazione statica che aiuta a prevenire diversi errori/bug, ma in questo linguaggio non sempre si deve dichiarare esplicitamente il tipo di dato ad una variabile, perché in molti casi sarà il compilatore stesso a dedurli, velocizzando cosi la scrittura, ma restando sempre ben sicuri. Nim è principalmente un linguaggio procedurale con supporto (minimo) ad OOP (ereditarietà, polimorfismo, invio dinamico) ma di suo non impone uno stile definito. Una caratteristica secondo me un pò controversa invece è che Nim non fa distinzioni tra maiuscolo e maiuscolo e tra *CamelCase* o *snake_case*, ovvero: "*ciaoMondo*" è uguale a "*ciao_mondo*" ma anche a *"cIAO_Mondo*" ecc.. e francamente questo potrebbe creare confusione, unica eccezione è la distinzione della prima lettera: "*Linguaggionim*" è diverso da "*linguaggionim*", ma questo (che vedremo nella sezione dello stile di Nim) serve per distinguere se la variabile è un tipo definito dall'utente o una variabile generica.

1.1 – Lo Stile di Nim.

Nella sezione "1" cercherò di scrivere tutti i preamboli e la parte più noiosa, cosi che si possa saltare a piedi pari per andare direttamente al sodo, o per recuperarle velocemente in caso di panico. Parlando qui di stile non è necessario seguirlo alla lettera, ma avere un modo più o meno condiviso da tutti per scrivere un programma può solo che semplificare la vita a tutti, specie se un domani scriverai delle tue librerie da condividere con il mondo.

Cominciamo vedendo la convenzione sulle righe:

- Le righe non dovrebbero mai superare gli ottanta caratteri per rendere il codice più leggibile.
- I rientri dei blocchi devono essere fatti con due spazzi bianchi (non è consentito l'uso del Tab!)
- Riflettere bene sull'uso degli spazzi bianchi (tranne quello detto sopra) perché alcuni editor potrebbero poi fare dei pasticci.

Vediamo qui la convenzione sui nomi dei tipi e delle variabili:

- Gli identificatori di tipo devono usare il PascalCase (*type MioTipo*).
- Gli identificatori di variabili devono usare il camelCase (*var miaVariabile*).
- Gli identificatori di variabili devono usare il snake_case (*var mia_variabile* non consigliata!).
- Gli identificatori di costante possono usare entrambi gli stili, ma se non strettamente necessario è meglio usare il camelCase (*const due = 1+1*).
- Le costanti che provengono da wrapper C/C++ il modo TUTTO_GRANDE è consentito ma è davvero brutto (meglio comunque usare il camelCase prima visto).
- Quando usiamo tipi "puntatori, al nome aggiungiamo il suffisso "Obj" o "Ref" o "Prt" rende più evidente cosa abbiamo tra le mani (*type NuovotipoRef = ref qualcosa*).
- Le varie eccezioni (exception) ed errori vari, dovrebbero avere il suffisso "Error" o "Defect" .

Convenzione di codifica:

- L'istruzione *return* dovrebbe essere utilizzata solo quando serve per controllare il flusso ad esempio quando ci son più possibilità di ritornare un valore magari a seguito degli *if* ma normalmente è meglio usare *result* che vedreai meglio in seguito.
- Cerca di usare il più possibile *prc* ed usare le funzionalità avanzate come le macro e template solo quando necessario.
- Utilizza l'istruzione *let* dove sei sicuro che i dati di quella variabile non cambiano, evitando di usare *var*, perché cosi sei sicuro che quel dato non potrà mai cambiare, ed il compilatore ottimizzerà meglio il tuo codice.

Convenzione su istruzioni a più righe:

- Le tuple su più linee dovrebbero avere i parametri allineati sulle righe.
- Allo stesso modo i parametri delle procedure dovrebbero stare allineati sulle righe.

Punteggiatura:

- dopo gli operatori :, ,, = ci va uno spazio esempio: *proc test(a: int, b: float) =*.
- Non occorre invece dopo l'operatore $ esempio: *echo("scrivo.." , $refVar)*.

Anche se nelle convenzioni prima citate ho affermato di lasciare solo due spazi di identazione, in questo manuale per maggior chiarezza, ne farò quattro ma solo per evidenziare il rientro che altrimenti potrebbe essere faticoso da vedere.

2 – Si Comincia.

Lasciata alle spalle la parte noiosa, ora possiamo cominciare a divertirci un pò e come iniziare se non nel modo più classico con il solito programmino "Ciao Mondo!!" che però ci darà già molte cose da vedere. Cominciamo creando un file "*ciao.nim*" sul nostro pc, poi lo apriamo con il nostro editor preferito e ci scriviamo:

```
echo("Ciao Mondo!!")
```

L'istruzione *echo* fa si che quanto scritto di seguito venga stampato a video, ma per poterlo eseguire bisogna compilarlo, quindi apriamo un terminale e scriviamo **nim c -r ciao.nim**. Il parametro **c** dice di usare il compilatore C (quindi il codice Nim verrà prima compilato in C) mentre il parametro **-r** dice di lanciare subito il nostro programma, altrimenti bisogna lanciarlo a mano. Ora prova ad aggiungere al tuo file le seguenti linee (in verde le aggiunte):

```
echo("Ciao Mondo!!")
echo "Ciao Mondo!!"
eChO("Ciao Mondo!!)
stdout.write("Ciao Mondo!!)
```

Compiliamo nuovamente il codice ed eseguiamolo, possiamo subito osservare almeno tre cose:

- In Nim le parentesi nelle funzioni possono essere omesse.
- Il nome della funzione non è sensibile allo stile di scrittura (**prima lettera esclusa!!**).
- La funzione *stdout.write()* è simile ad *echo* stampa a video una stringa, ma non va alla riga successiva automaticamente.

Un'altra funzione interessante per iniziare, imparentata con *echo*, è quella che consente di inserire i dati da tastiera ovvero *readLine*, un input insomma, vediamo un esempio creiamo un file di nome "*input.nim*" e scriviamo:

```
stdout.write("Inserisci Una Stringa: ")
let str = stdin.readLine()
echo("Hai inserito: ", str)
```

Compilalo come prima con **nim c -r input.nim** e vedrai sulla console una frase che ti richiede qualcosa e di fianco potrai inserire una stringa, fatto questo e premuto invio apparirà quanto inserito. Per ora non preoccuparti se non è tutto chiaro, perché verrà ripreso e spiegato tutto più avanti ora è solo per prendere un po' di confidenza con la compilazione, come scrivere e rompere un po' il ghiaccio.

2.1 – Assegnazione Delle Variabili.

In un linguaggio fortemente tipizzato come Nim, l'assegnazione delle variabili è fondamentale, ma prima di indicare il tipo di variabile (char, int, string…) bisogna dire al compilatore se quella variabile è mutabile o altro, ci sono tre grosse distinzioni:

var**:* una variabile dichiarata come *var* può variare il suo valore (ma non il *tipo*, se è ***int deve rimanere int) esempio: *var numero = 3* in questo caso la variabile vale inizialmente 3 (intero) ma le si può riassegnare un valore: *numero = 7* e sarà un operazione lecita.
let: una variabile dichiarata come *let* non può piu variare il suo valore durante l'esecuzione, se lo si tenta di fare il compilatore darà un errore: *let numero = 3* se ora tentiamo di fare *numero = 5* la compilazione fallirà (cannot be assigned to).
const: una variabile dichiarata come *const* sarà pure lei immutabile, ma ha una particolarità; il suo valore deve essere calcolabile in fase di compilazione, vediamo un esempio:

```
proc summ():int=
   for a in 1..10:
      result = result + a

const somma = summ()
```

Anche se per ora non sappiano come funzionano le procedure ed il resto, l'importante è capire che a *somma* verrà assegnato in fase di compilazione il numero 55 perché gia in quella fase è possibile calcolarlo e renderà il tuo programma più veloce in fase di esecuzione. Una volta deciso se una variabile è mutabile o altro, va assegnato il tipo a cui fa capo. I tipi base di Nim sono i seguenti:

- Booleni, *bool* che possono avere solo due valori: *true* o *false*.
- Carattere, *char* sono dei caratteri in formato ascii: 'a'.
- Stringa, *string* sono una serie di caratteri solitamente in formato unicode: "Ciao Mondo!!".
- Interi, *int* sono i numeri senza decimali, però qui ci sono diverse sotto tiplogie:
  - *int* numero intero con numero di bit variabile dalla piattaforma con segno (default).
  - *int8* numero intero ad 8bit con segno.
  - *int16* numero intero a 16bit con segno.
  - *int32* numero intero a 32bit con segno.
  - *int64* numero intero a 64bit con segno.
  - *uint* numero intero con numero di bit variabile dalla piattaforma senza segno.
  - *uint8* numero intero ad 8bit senza segno.
  - *uint16* numero intero a 16bit senza segno.
  - *uint32* numero intero a 32bit senza segno.
  - *uint64* numero intero a 64bit senza segno.
- Decimali, *float* sono i numeri con la parte decimale, anche essi hanno varie tipologie:
  - *float* numero decimale con numero di bit di default a 64bit.
  - *float32* numero decimale con 32bit.
  - *float64* numero decimale con 64bit (dichiarato).

Potrebbe tornarti utile, specie se lavori su microcontrollori poter ad esempio variare il tipo da *int* a *int8* ovviamente questo è possibile farlo e vedrai ora due modiper farlo:

```
let variabile: int = 75                    → type(variabile) = int
let cambio = int8(variabile)               → type(cambio) = int8; fatto cast da int a int8.
let cambioUnsafe = cast[int16](variabile)  → type(cambioUnsafe) = int16 metodo unsafe PERICOLO!! Cast da int a int16.
```

Vedrai ora degli esempi di assegnazione di vario genere:

```
let intero = 75
let esadecimale = 0x75
let ottale = 0o75
let binario = 0b1001011
echo(type(intero))                         → int
```

Il tipo non è esplicitamente dichiarato, questo perchè il compilatore Nim riesce a dedurlo da solo in base a la valore specificato, chiaramente se non assegni nessun valore alla variabile il compilatore non potrà indovinarlo e quindi va esplicitamente dichiarato. Nel caso *let intero = 75* verrà assegnato in automatico il tipo *int*, nel caso lo volessi dichiarare tu stesso un altro tipo, allora devi fare:

```
let intero: int16 = 75
let intero = 75 'i16
type(intero)                               → int16
```

Puoi usare entrambe le diciture per dichiarare la variabile ma la prima è di certo la più chiara e semplice da ricordare.

> **type(): la procedura type ti salverà molte volte, perché se non sai che tipo ritorna una funzione o ti che tipo è una variabile, con la procedura type() viene presto svelata.**

3 – I Tipi di Dati.

Ora che hai visto come dichiarare le variabili ed i tipi base di Nim, credo si giunta l'ora di vedere le strutture dati presenti, sia nella libreria standard che in alcune librerie presenti di default e magari le procedure che ci aiutano a gestirle al meglio, ricordo che Nim segue la filosofia "batterie incluse" quindi moltissime cose son già lì a nostra disposizione.

3.1 – I Numeri.

Iniziamo dai numeri che si useranno spesso e volentieri. Di default gli operatori sui numeri son grossomodo quelli classici usati in tutti i linguaggi:

```
echo(5 / 2)          → 2.5
echo(5 div 2)        → 2
echo(2 mod 2)        → 0
echo(5 + 2)          → 7
echo(5 - 2)          → 3
echo(16 shr 2)       → 4  o in altre parole 00010000b sposto il bit che vale 16 di 2
                        posizioni 00000100b.
echo(4 shl 2)        → 16
echo(toInt(0.6))     → 1
echo(toInt(1.4))     → 1
echo(toFloat(2))     → 2.0
echo(0b0111 and 0b0101)  → 5  ovvero 0b0101 che è l' and logico.
```

Come puoi vedere non son complessi e dall'esempio sopra risulta chiara la loro funzione, ma andrò a spiegarli comunque:

- /: operatore di divisione e ritorna un numero float.
- *: moltiplicazione.
- ***div***: operatore di divisione e ritorna un numero intero (tronca i decimali senza arrotondare).
- ***mod***: operatore modulo ritorna zero se i numeri son divisibili tra loro.
- +: operatore somma, addiziona due o più numeri tra di loro.
- -: operatore sottrazione, sottrae un numero all'altro.
- ***shr***: operatore che sposta i bit a destra, nell'esempio sarebbe: 00010000b → 00000100b.
- ***shl***: operatore che sposta i bit a sinistra, nell'esempio sarebbe: 00000100b → 00010000b.
- ***toInt***: metodo che arrotonda un numero float all'intero più prossimo (3.14 → 3; 1.99 → 2).
- ***toFloat***: metodo che converte un intero in un float.
- ***and, or, xor, not***: calcolano il valore bit per bit dei numeri dati.

In più avrai a disposizione il modulo ***math*** (https://nim-lang.org/docs/math.html) che ti mette a disposizione parecchie funzioni matematiche, vediamo qualche esempio:

```
proc sqrt(x: float64): float64 {...}
echo(sqrt(16.0)) → 4.0 (nota che la procedura vuole un numero float!!)
```
Procedura che ritorna la radice quadrata di un numero float.

```
proc pow(x, y: float64): float64 {...}
echo(pow(3.0, 6.0)) → 729
```
Procedura che ritorna la potenza tra due numeri float.

```
proc round(x: float64): float64 {...}
echo(round(5.2)) → 5.0
echo(round(5.7)) → 6.0
```
Procedura che arrotonda al valore più prossimo a quello passato in argomento. Nelle versioni Nim fino a 1.6.x almeno compila con ***-d:nimPreviewFloatRoundtrip*** per avere l'arrotondamento corretto!!

Ti ho mostrato solo un lieve assaggio di tutte procedure presenti, ma ora sai che quando avrai a che fare con operatori matematici, questo è il primo posto da visitare per vedere se già esiste quello che ti serve per risolvere il tuo problema. Altri moduli che contengono funzioni matematiche sono ***complex*** (https://nim-lang.org/docs/complex.html) e ***rationals*** (https://nim-lang.org/docs/rationals.html) ma che forse non utilizzerai mai, quindi mi sembra superfluo dargli un occhiata in questa sede.

3.2 – Le Stringhe ed i Caratteri.

Le stringhe sono una serie di lettere di solito in unicode mentre i caratteri son semplici caratteri in ascii. Vediamo con qualche esempio cosa si può fare:

```
let str1 = "Ciao "
let str2 = "Mondo!"
let char1= 'c'
var str3 = "Nim "

echo("Unione: ", str1 & str2)                  → Unione: Ciao Mondo!
echo("str2 contiene ", len(str2), " lettere")  → str2 contiene 7 lettere
echo("Valore di c: ", ord(c))                  → Valore di c: 113
echo("Il Numero 123 (0x7b) equivale a: ", chr(123))  → Il Numero 123 (0x7b) equivale a: {
str3.add("è Bellissimo!!)
echo("Aggiungo una stringa: ", d)              → Aggiungo una stringa: Nim è Bellissimo
```

Qui ho messo un bel po' di carne al fuoco, nelle prime tre righe, ho impostato come *let str1, str2, char1* questo perchè non devono mai mutare nel programma, mentre *str3* dove andrò ad aggiungere qualcosa, dovrà essere di tipo *var*. Ma le cose più importante sono quelle in rosso, che andrò ad analizzare:

- ***&***: operatore che concatena due stringhe tra di loro: "ciao "&"mondo" = ciao mondo.
- ***len***: operatore che ritorna la lunghezza della stringa (o di una sequenza etc): len("albero") = 6.
- ***ord***: operatore che ritorna il valore ascii del carattere messo in argomento: ord('c') = 113.
- ***chr***: operatore che ritorna il carattere del equivalente numero ascii: chr(123) = {.
- ***add***: operatore che aggiunge una stringa o un carattere ad una stringa.

Apparentemente sembrano già molte cose, ma è solo l'inizio, perchè sulle stringhe, i vari moduli hanno ancora molte sorprese da offrirti, è ovvio che non posso elencare tutto, ma almeno le cose più interessanti si. Cominciamo dal modulo ***strutils*** (https://nim-lang.org/docs/strutils.html) che come potrai intuire offre diverse utility che vanno ad operare sulle stringhe vediamo alcune procedure di questo modulo:

```
proc strip(s: string; leading = true; trailing = true;chars: set[char] = whitespace): string
```

Questa procedura elimina dei caratteri, di default gli spazi vuoti all'inizio ed alla fine, ma variando i parametri si possono ad esempio anche eliminare delle lettere ben specifiche.

```
proc split(s: string; sep: char; maxsplit: int = -1): seq[string]
let str1 = ("1, 2, 3, 4); let sp = split(str1,',',2)        → @["1", "2", "3", "4"]
```

Procedura utilissima per separare da una stringa gli elementi per poi metterli in una sequenza ed accedevi comodamente.

```
proc isUpperAscii(c: char): bool
```

Controlla se un carattere ascii è minuscolo o maiuscolo, ovviamente ce la controparte *isLowerAscii*.

```
proc intToStr(x: int; minchars: Positive = 1): string
```

Procedura per convertire un numero intero in una stringa sempre in questa categoria poi ce *toHex* e *toBin*.

```
proc trimZeros ( x : var stringa ; decimalSep = '.' )
var num = "123.9870000"; num.trimZero('.')        → 123.987
```

Elimina gli zeri dai numeri formattati come stringhe (non ritorna una copia).

```
proc parseInt(s: string): int
```

Questa assieme a *parseFloat* ed altre sono molto utili per convertire dati passati sotto forma di stringa in dati "numerici", ad esempio quando usiamo *readLine* che accetta solo stringhe, usando queste funzioni convertiamo il suo ingresso in un numero intero o float o altro. Ti invito caldamente a dare un occhiata a questa libreria che è molto vasta e contiene molte utility ed interessanti procedure che potrebbero tornarti molto utili, ma non è finita, perché ce un' altro modulo interessante che opera sulle stringhe ed è ***strformat*** che serve a fare ciò che fanno le "f-string" di Python.

```
import strformat

let mod = "Punto"
let eta = "6"

echo("La mia automobile ", mod, " ha ", eta, " anni")      → La mia automobile Punto ha 6 anni
echo(fmt("La mia automobile {mod} ha {eta} anni"))         → La mia automobile Punto ha 6 anni
```

Il risultato finale è lo stesso, ma mentre nel primo echo si deve continuamente aprire e chiudere virgolette e inserire le virgole, nel secondo la scrittura è molto più naturale e semplice e per far ciò si usa *fmt*, ma non è tutto perchè ci permette anche di formattare i numeri come meglio vogliamo ad esempio:

```
echo(fmt("{12: 04}"))          → 0012 ovvero scriverà il 12 ma preceduto da due zeri che occuperanno le 4 posizioni.
echo(fmt("{1.2345: 0.3f}")     → 1.234 elimina tutti i decimali oltre la terza posizione.
```

Attenzione!! Quanto visto sopra non va a modificare la variabile, ma è solamente un rappresentazione manipolata di essa.

3.3 – Gli Array.

Gli array, o matrici sono dei semplici contenitori di lunghezza fissa, ovvero una volta dichiarato non si può ne levare ne aggiungere elementi ma solo modificare quelli esistenti e tutti gli elementi devono essere dello stesso tipo. Qui di seguito vedrai due modi per dichiararli che commenterò dopo:

```
let matrice = [1, 2, 3, 4 ,5]                  → inizializza un array di cinque elementi int (indice da 0 a 4).
var matrice2 = [uint8(6), uint8(34), unit8(3)] → inizializza un array di due elementi luint8l.
echo(matrice)                                  → [1,2,3,4,5]
echo(matrice[2])                               → 3
let matrice3: array[0..2, int] =  [1, 2, 3]    → inizializza un array di tre elementi interi con indice da 0 a 2.
let matrice3: array[5..7, int] =  [1, 2, 3]    → inizializza un array di tre elementi interi con indice da 5 a 7.
```

Nella prima linea assegni alla variabile *matrice* un array di cinque elementi di interi, nella seconda linea creiamo un array di due elementi ma di tipo ***uint8***, nella terza linea invece stampiamo tutto l'array. La quarta linea invece è più interessante perché introduce l'operatore "**[]**" che ci permette di andare a richiamare all'interno di un array un valore o come vedremo tra breve un range di valori. Nelle ultime due linee inizializziamo sempre un array di interi, ma stavolta lo dichiariamo esplicitamente come deve essere (tre elementi di interi) , in più questa forma di scrittura ti da la possibilità di stabilire gli indici dell'array. Vediamo alcuni esempi di come chiamare i valori contenuti negli array:

```
echo(matrice[1..3])    → [2, 3, 4]   ritorna l'intervallo dal secondo al quarto elemento.
echo(matrice[2..^1])   → [3, 4, 5]   ritorna l'intervallo dal terzo all'ultimo elemento.
echo(matrice[^1])      → 5           ritorna solo l'ultimo elemento.
echo(matrice[^2])      → 4           ritorna solo il penultimo elemento.
```

Qui introduciamo due nuovi operatori, ovvero "**..**" e "^". Il primo ti consente di fare un slice tra vari elementi dell'array, ovvero estrarre dell'array gli elementi indicati dagli indici, per esempio (0..3 → 1, 2, 3 ,4) mentre il secondo è come contasse a ritroso la posizione (^1 = ultimo, ^2 = penultimo e così via..). L'operatore "**..**" lo incontrerai ancora molte volte in futuro e sarà sempre molto utile. Ma come si contano gli elementi in un array? Normalmente il primo valore equivale all'indice zero, quindi se vuoi ottenere il primo valore di un array dovrai chiamare il valore in posizione zero, quindi l'ultimo sarà in posizione "**elementiarray -1**" (per fortuna l'operatore "^" sa benissimo questa cosa). Se vuoi cambiare questi indici, e farlo partire da 5 anzichè

da 0, ti basta dichiararlo come nell'esempio a pagina 9 o qui sotto scritto in color indaco. Vediamo il secondo modo di dichiarazione di un array;

```
type
      Matrice = array[0..4, int]
      Matrice2 = array[1..5, int]
let Matrice = [1, 2, 3, 4, 5]
let Matrice2 = [1, 2, 3, 4, 5]
echo(Matrice[1])                    → 2
echo(Matrice2[1])                   → 1
```

In effetti questo modo di dichiarare l'array è un po' strano, perché andrai a creare un tipo di dato nostro (vedremo dopo il type), e nella dichiarazione dirai solo che vuoi un array di cinque elementi interi (nel nostro caso) che per ora è vuoto e solo in un secondo momento lo riempirai. Cosa importante però è la scrittura "0..4 o 1..5", perchè non andrà a riempire l'array con quei valori, ma creerà gli indici a partire da quei valori, che possono essere qualunque numero (intero) per fino negativi, ecco perché in *Matrice* l'elemento 1 è 2, mentre in *Matrice2* l'elemento 1 è 1.

3.4 – Le Tuple

Le ***tuple*** sono fortemente imparentate con gli array che hai appena visto, cioè una volta dichiarati non si possono più aggiungere ne levare elementi, ma solo modificare quelli esistenti (se dichiarato come *var*) però a loro differenza che devono essere omogenei, le ***tuple*** possono contenere *tipi* diversi, ecco un esempio:

```
var tup = ("ciao", 123, 45.89, 'o')
echo tup[0]                         → ciao
tup[0] = "Nim"
echo tup                            → ("Nim", 123, 45.89, 'o')
```

Questa struttura dati, può essere molto utile, specialmente quando userai le procedure e dovrai tornare più valori contemporaneamente e magari di tipi non omogenei, le ***tuple*** ti saranno di grande aiuto, specialmente nella loro versione ***nameTuple***, che differisce da quanto vista prima, perchè gli elementi hanno un nome e lo potrei utilizzare per richiamare il valore ad esso associato senza usare gli indici :

```
var tup = (a1: "ciao", a2: 123, a3: 45.89, a4: 'o')
echo tup.a1                         → ciao
tup.a1 = "Nim"
echo tup                            → ("Nim", 123, 45.89, 'o')
```

Vero, il risultato non cambia da come scritto prima, ma la comodità di richiamare i valori in questo modo, è innegabile.

3.5 – Le Sequenze.

Le sequenze sono simili agli array, contengono dei dati di tipologia uniforme, ma differiscono per due cose: la prima molto importante è che le sequenze possono modificarsi durante l'esecuzione del tuo programma, ovvero potrai inserire e levare degli elementi. La seconda è che non è "re-indicizzabile", cioè il primo elemento sarà sempre l'elemento di indice zero. Vedrai ora come inizializzarle e gestirle:

```
var seq1 = newSeq[int](3)           → inizializza una sequenza vuota di interi (costruttore)  → @[0, 0, 0].
var seq2 = @[1, 2, 3, 4]            → inizializza una sequenza di interi.
var seq3: seq[int]                  → inizializza una sequenza vuota di interi.
var seq4 = @[uint16(60), uint16(45)]  → inizializza una sequenza di due elementi di tipo uint16.
var seq5 = newSeqofCap[int](2)      → crea una sequenza vuota len() = 0, ma verrà riallocata solo se si supera il
                                      numero di elementi contenuti/aggiunti indicato non prima di esso.

type
      seq4 = seq[int]               → inizializza un tipo sequenza di interi.
```

Qui ti ho messo parecchie cose per avere un quadro completo, ma le più importanti secondo me sono la seconda e la terza riga; nella seconda viene costruita già da subito una sequenza di quattro elementi di interi, nella terza si inizializza una sequenza di interi, ma tutta ancora da riempire. Come prima, i *type* l'ho messo ma lo spiegerò dopo, ora è solo per avere da subito il quadro della situazione. A questo punto però, vediamo come si gestiscono queste sequenze e ci saranno degli operatori appositi:

```
seq2 .setLen(3)          → @[0, 0, 0]
seq2.len()               → 3
seq2.add(75)             → @[0, 0, 0, 75]
seq2.insert(8,3)         → @[0, 0, 0, 8, 75]
seq2.delete(3)           → @[0, 0, 0, 75] preserva l'ordine degli elementi in memoria; più lento!!!
seq.del(3)               → @[0, 0, 0] non preserva l'ordine degli elementi in memoria; più veloce!!
echo(seq2.pop())         → @[0, 0] e stampa lo 0 (zero) rimosso.
seq1 & seq2              → @[1, 2, 3, 4, 0, 0]
```

Per la gestione delle sequenze abbiamo molti operatori che ci vengono in aiuto. Ora li spiego meglio ma ci saranno anche vecchie conoscenze.

- ***setLen***: riempirà di zeri la sequenza, quanto l'argomento passato, nel nostro caso tre zeri.
- ***len***: ritornerà la dimensione (quanti elementi contiene) la nostra sequenza.
- ***add***: va ad aggiungere un elemento alla sequenza.
- ***insert***: accetta due parametri, il primo è l'elemento da inserire, il secondo la sua posizione nella sequenza.
- ***delete***: elimina il parametro nell'indice desiderato, e preserva l'ordine degli elementi in memoria (lento).
- ***del***: elimina il parametro nell'indice desiderato, ma non preserva l'ordine degli elementi in memoria (veloce).
- ***pop***: elimina l'ultimo elemento nella sequenza e lo rende subito stampabile a video.
- ***&***: se usato per due o più sequenze, le concatena creandone una sola.
- ***c[x..y]***: slice della sequenza intera tra i due indici x e y compresi.
- ***c[x..^y]***: slice degli indici tra x y è conteggiato a partire dalla fine (ultimo e = ^1 penultimo =^2 ecc).
- ***c[x..<y]***: slice degli indici tra x e y-1; l'operatore "<" esclude (scala di uno) il valore indicato.

ATTENZIONE!! le sequenze lavorando sulla heap quindi son più lente rispetto agli array che son allocati nello stack, se non pensi di doverle modificare ma di lasciarle costanti, usa sempre gli array, aumenti la velocità del tuo programma.
Se sai per certo quanti elementi massimi saranno inseriti nella sequenza, usa pure *newSeqOfCap* che preparerà la memoria per contenerei dati senza dover ri-alloccare continuamente come invece fa *newSeq* questo aumenterà la velocità del tuo programma, ovvio che puoi superare quanto imposto, ma il vantaggio prima citato sarà vanificato.

3.6 – Tables.

Le ***tables***, sono delle tabelle hash estremamente efficienti, che in altri linguaggi, come Python vengono chiamate dizionari, e per usarle dovrai importare il modulo ***tables***. Ogni voce è costituita da un coppia chiave-valore. Questo ti fa intuire che grazie alla chiave sarà molto facile trovare un valore, e come prima accennato le rende estremamente veloci. Ecco un esempio di come si può usarle:

```
import std/[tables]

var agenda = initTable[string, int]()
echo(agenda)                          → {:} la tabella ora è ancora vuota
agenda["Susanna"] = 123456
agenda["Luca"] = 654321
```

```
echo(agenda)                          → {“Susanna”: 123456, “Luca”: 654321} stampa tutto il contenuto di agenda
echo(agenda[“Susanna”])               → 123456 stampa il numero di telefono di Susanna
```

Dall’esempio appare chiaro come è facile trovare gli elementi o valori dentro a questo tipo di tabelle. Si può anche crearle con già dei valori esistenti facendo ad esempio:

```
var agenda2 = toTable({"Franco": 999999, "Giulio": 333333, "Maria": 444444})
var agenda3 = {"Silvia": 777777, "Luigi": 55555}.toTable
```

In questo tipo di tabelle generalmente non interessa mantenere traccia dell’ordine di inserimento, anche perché questo influirebbe sulla loro efficienza, ma potrebbe capitare per qualche motivo o per poter usufruire dei metodi specifici) che questo ordine sia essenziale anche a costo dell’efficienza allora potrai creare la tabella nel seguente modo:

```
var agenda4 = toOrderedTable({"Lola": 782345, "Enrico": 339922})
var agenda5 = {"Alberto": 0012203, "Lorena": 727374}.toOrderedTable
```

Un’altra funzionalità di questo tipo di tabelle, è che possono contare automaticamente gli elementi uguali, che compaiono in una stringa, in una sequenza o in un array, ecco come:

```
const frase = "tabella bella"
let frequenza = toCountTable(frase)
echo frequenza                        → {‘a’:3, ‘t’: 1, ‘l’: 4, ‘b’:2 ,’ ‘:1, ‘e’:2}
```

Questi sono i tre modi per creare delle hashtables (dizionari). Ora come di consueto vedrai alcuni metodi per poter interagire con loro:

- ***add***: aggiunge alla tabella una nuova voce (chiave-valore).
- ***clear***: cancella tutti i dati contenuti nella tabella.
- ***contains***: controlla se una chiave è contenuta nella tabella.
- ***del***: cancella la voce con la chiave passata come argomento.
- ***getOrDefault***: recupera il valore associato alla chiava passata se non ce torna un valore predefinito.
- ***hasKey***: ritorna “true” se la chiave cercata è nella tabella.
- ***len***: come già sai ritorna il numero di elementi (chiave-valore) presenti in tabella.
- ***pop***: elimina la chiave indicata dalla tabella e se esistente torna “true” e può trasferire il valore ad una variabile passata in argomento.

3.7 – i Set.

I set sono degli insiemi e rispondono alla matematica di essi (intersezioni, unioni, sottoinsiemi, ecc) ma la cosa da tenere sempre ben a mente è che in un set, gli elementi contenuti sono sempre univoci (non possono ripetersi). Nei set della libreria di sistema si possono usare solo tipi ordinali, ovvero elementi in cui la successione è nota: 2 viene dopo di 1 ma prima di 3, ‘b’ viene dopo ‘a’ ma prima di ‘c’ e cosi via notare l’uso di ‘’ che indica che stiamo parlando di caratteri e non stringhe!!, una stringa per esempio non è ordinale. Anche i tipi che puoi utilizzare con questi insieme sono limitati (vedi sotto) perché cosi si garantisce l’alta efficienza, ma vedrai che ci sarà modo di evitare queste limitazioni, a scapito però della performance. Passiamo a vedere l’inizializzazione dei set.

```
var setInt1 : set[int16]              → inizializza un nuovo set di interi a 16 bit vuoto.
var setInt2 = {1,2,3,4,2}             → inizializza un nuovo set di interi e già si assegnano i valori.
var setInt3 = {3,4,5,6}               → set  in più per gli esempi delle operazioni su di essi.

type
      SetChar3 = set[char]            → inizializza un tipo di set che accetta caratteri.
```

I tipi usabili sono: int8-int16, unint8/bitye-uint16, char, enum.

Altra cosa molto importante è che i set non sono un insieme ordinato, quindi non si può in nessun caso richiamare gli elementi tramite un indice, ma sono estremamente utili oltre che per eliminare i doppioni come vedremo per verificare la presenza di un elemento con l'operatore "**in**". Anche nei set si può aggiungere o eliminare dei valori.

```
echo(setInt2)                  → {1,2,3,4} come puoi notare set ha rimosso 2 in quanto doppio.
setInt1.incl(75)               → {75} hai appena aggiunto il numero 75 al set "setInc1".
SetInt1 + setInt2              → {1,2,3,4,5,6} nota la mancanza dei doppi anche qui.
SetInt1 * setInt2              → {3,4}
setInt1 - setInt2              → {1,2}   ATTENZIONE!!
setInt2 - setInt1              → {5,6}   ATTENZIONE!!
setInt1 < setInt2              → false.
3 in setInt1                   → true.
```

Anche per i set ci sono molti interessanti operatori che ora cercherò di spiegarli meglio:

- ***incl***: andrà ad inserire un elemento (di quelli concessi) dentro al set, accetta solo un elemento per volta.
- +: **unione**, unisce assieme due set, ovviamente escludendo eventuali doppi elementi.
- *: **intersezione**, ritorna solamente gli elementi in comune dei due set.
- -: **differenza**, dal primo insieme indicato, sottrae (elimina) gli elementi che sono in comune al secondo insieme. Prestare attenzione quindi all'ordine con cui vengo scritti!!
- <: **sottoinsieme**, se il primo set è sotto insieme del secondo ritorna true. Es {3,4} < {1,2,3,4,5} = true.
- ***in***:**è contenuto in**, questo operatore lo useremo molto spesso, su set, sequenze array, perché offre una ricerca rapida di un elemento all'interno di queste strutture dati (molto veloce).

Come vedi i set son molto semplici ma molto efficienti, a volte però magari vogliamo o dobbiamo lavorare con altri tipi di dati, dati non supportati da questi insiemi. Niente paura, Nim ha già pronto il modulo "***sets***" che risolve questo problema e potremmo inserire qualsiasi "cosa" a cui si può applicare un *hasing*. Non mi soffermo su come lavorano, perché è identico (o quasi vedremo dove ce qualche differenza) a quanto visto per i set di sistema, ma cambia solo qualcosa sulle inizializzazioni e i metodi/operatori. Al solito vediamo come si inizializzano.

```
var hSet1: HashSet[float]                  → inizializza  un nuovo hashset vuoto di tipo float (ora si può!).
var hSet2: HashSet[int] = toHasSet([2,4])  → inizializza un nuovo hashset di tipo int e mettiamo dei valori.
```

In questo modulo però, i set contrariamente a quanto nella libreria di sistema, possono anche essere di tipo ordinato o meglio si ricorda l'ordine di inserzione degli elementi.

```
Var hSet3: OrderSet[string]                       → inizializza un nuovo hashset ordinato di  tipo string vuoto.
Var hSet4: Oredeset[string] = toOrderSet(["ciao","Nim"])  → inizializza un nuovo hashset ordinato con dei valori.
```

Tutti i metodi che hai visto prima per i set "normali" valgono anche qui (ci sono degli alias però tipo – alis ***difference***) ma alcuni operatori potrebbero essere interessanti da vedere e spiegare, per gli altri dai un occhiata qui https://nim-lang.org/docs/sets.html .

- ***toHashSet***:inserisce una struttura dati (array, set, sequenza) in un hashset.
- ***toOrderSet***: inserisce una struttura dati (array, set, sequenza) in un orderhashset.
- ***clear***: elimina tutti i dati contenuti nel hashset.
- -+-: differenza simmetrica tra due insiemi, elimina gli elemnti in comune e ritorna solo quelli diversi su entrambi gli insiemi: {1,2,3,10} -+- {1,2,3,4,7} = {4,10,7}

3.8 – Gli Ordinali.

Ho già accennato agli ordinali nel paragrafo 3.5 parlando dei set, ma qui li riprendo un po' più nello specifico. Sono tutti quei elementi in cui esiste una successione nota; ad esempio i numeri interi che sono "in ordine" 1,2,3… ma possono esserlo anche i caratteri che appunto seguono l'ordine alfabetico 'a', 'b', 'c'…. i booleniani false (0) true (1) egli enum (enumerazioni). Vediamo qui quali operatori possono agire sui ordinali.

```
var esempio = 2          → prendiamo la variabile "esempio" che vale 2.
let array = [0, 1, 2, 3] → creiamo un array di quattro elementi.
succ(esempio)            → 3 attenzione però, esempio = 2 ancora!!!!! (non agisce sulla variabile).
inc(esempio)             → 3 questa volta è stata incrementata la variabile "esempio".
high(array)              → 4
ord('N')                 → 78
```

Per intero gli operatori disponibili:

- ***succ***: ritorna il valore successivo a quello dato come argomento, senza incrementare il valore della variabile passata.
- ***pred***: ritorna il valore precedente a quello dato come argomento, senza decrementare il valore della variabile passata.
- ***inc***: incrementa il valore della variabile passata.
- ***dec***: decrementa il valore della variabile passata.
- ***high***: ha più funzioni se usato come high(int) torna il valore massimo che int può contenere, ma più importante è l'uso con gli array perché questo comando ti ritorna il valore dell'indice più alto. Nelle sequenze non si usa, perché in esse l'indice minore parte sempre da zero (e si usa let) quindi non ce possibilità di errore, mentre negli array gli indici possono partire arbitrariamente.
- ***low***: ha più funzioni se usato come low(int) torna il valore minimo che int può contenere. Se usato con gli array ritorna il valore dell'indice più basso (potrebbe essere negativo o comunque diverso da zero).
- ***ord***: ritorna il valore intero dell'ordinale passato come argomento.

3.9 – Varie.

Ora che abbiamo visto un po' di cose, in questa sezione varie, cercheremo di metterle un po' a frutto con qualche esempio che commenterò alla fine. Non preoccuparti se ancora delle cose non le capirai, le vedremo nella prossima sezione, concentrati solo sui tipi di dati e sui vari operatori che userò. Il programma che vedremo non avrà molta utilità, ma sarà funzionale e spero faccia capire come usare alcune delle cose che hai visto prima.

```
1 import std/[strformat,sets]

2 let ar1 = [1, 5, 9, 8, -34, 2]
3 var se1 = @[27, 65, 20, -187, 90]

4 echo (fmt"array1 = {ar1} sequenza1 = {se1} e contiene {len(se1)} elementi")
→ array1 = [1, 5, 9, 8, -34, 2] sequenza1 = @[27, 65, 20, -187, 90] e contiene 5 elementi

5 if 55 notin se1:
6     se1.add(75)
7     se1.add(90)
```

```
8  echo (fmt"array1 = {ar1} sequenza1 = {se1} e contiene {len(se1)} elementi")
→ array1 = [1, 5, 9, 8, -34, 2] sequenza1 = @[27, 65, 20, -187, 90, 75, 90] e contiene 7 elementi

9  var set1 = toHashSet(se1)
10 echo (fmt"array1 = {ar1} hashset = {set1} e contiene {len(set1)} elementi")
→ array1 = [1, 5, 9, 8, -34, 2] hashset = {20, 27, 65, -187, 75, 90} e contiene 6 elementi
11 echo (fmt"Da elem 1 ->3 {se1[1..3]}")
Da elem 1 ->3 di sequenza se1@[65, 20, -187]
12 echo("Lista elementi array:")
13 for c in ar1.low..ar1.high:
14     stdout.write(ar1[c]," ")
→ Lista elementi array: 1 5 9 8 -34 2
15 echo('\n', "Lista elementi sequenza:")
16 for d in 0..<len(se1):
17     stdout.write(se1[d], " ")
→ Lista elementi sequenza: 27 65 20 -187 90 75 90
18 echo()
19 var `inc` = 0
20 for e in 0..5:
21     inc.inc
22     echo(fmt("Ora inceremento vale: {inc}"))
→ Ora inceremento vale: 1, Ora inceremento vale: 2, .. , Ora inceremento vale: 6
```

Nella prima riga, importi i moduli di cui avrai bisogno con ***import***, nota anche come è stato scritto ovvero ***std/[strformat, sets]*** questa dovrebbe essere la corretta forma per farlo (si legge dalla libreria standard, importa strformat, sets). Nella seconda e terza riga crei rispettivamente un array di sei elementi interi ed una sequenza di cinque elementi. Nella quarta riga stamperai sia l'array che la sequenza, sfruttando quella che in Python si chiama "f-string" in piu con l'operatore ***len*** ti dirà di quanti elementi è costituita. Nella quinta linea, fai un controllo con l'operatore ***notin*** per verificare se il numero 55 è presente nella lista, se non lo è lo andrà ad aggiungere con l'operatore ***add*** ed aggiungerà anche un'altro 90 (ora son due nuovi elementi aggiunti nella sequenza), poi ristampa la sequenza ed il numero degli elementi aggiornati. La nona riga converte (in una nuova variabile) la sequenza *se1* in un hasSet grazie all'operatore ***toHasSet,*** come noterai nella linea successiva gli elementi non saranno sette come in *se1*, ma saranno solo sei, perché se ti ricordi bene nei set non possono esserci due valori uguali (e ora avremo due 90!!). Nella linea undici, stamperai soltanto i valori della sequenza *se1* compresi tra l'indice 1 e 3. La linea tredici, crea un ciclo che va dall'indice inferiore (qual che sia) di *ar1* all'indice superiore e la variabile *c* assumerà il numero di indice, poi nella linea quattordici verrà stampato il valore di *ar1[c]* (lo si può fare in modo molto più semplice scrivendo *for c in ar1*: ma non raggiungerebbe lo scopo di illustrare questi operatori!). Nella linea sedici, crei un secondo *for* ma sta volta per la sequenza, e qui puoi osservare che non usiamo *low* e *high* che va bene per gli array, ma usiamo l'operatore ***<len()*** che è quello corretto per questo tipo di dati. Nella linea diciannove, potrai notare una strana dichiarazione della variabile, ovvero ***`inc`*** questo è detto *stropping*. Se ti ricordi, *inc* è una parola riservata di Nim ma usando l'accento capovolto (` = "AltGr + '") puoi usarla per dichiarare una variabile; certo se puoi evita di farlo ma a volte magari non si può. Alla linea venti crei un ciclo che andrà da 0 a 5 e nella linea ventuno incrementerai la variabile *inc* con l'operatore ***inc*** (vedi perché dico che è meglio evitarlo, è orrendo da vedere e da capire) e stampiamo a video il risultato (valore per valore). Per concludere ci sono ancora un po' di metodi che val la pena esaminare perché torneranno utili in futuro.

- ***is***: verifica se due argomenti son dello stesso tipo esempio: *let a =1; a is int* → true.
- ***isnot***: verifica se due argomenti non son dello stesso tipo.
- ***addr***: prende l'indirizzo di una locazione di memoria.
- ***A and B***: fa l'and logico (bit per bit) tra A e B.
- ***A or B***: fa l'or logico (bit per bit) tra A e B.
- ***A xor B***: fa lo xor logico (bit per bit) tra A e B.
- ***not A***: fa la negazione di A.

4 – Le strutture di Controllo.

Fino ad ora hai visto i vari tipi e le strutture dati che Nim può gestire direttamente, ma per scrivere un programma degno di questo nome, ci vuole almeno qualcosa che in qualche modo controlli/agisca su questi dati. Normalmente le istruzioni vengono eseguite una dopo l'altra ma con queste strutture possiamo variare questo ordine. Si può affermare dunque che qualsiasi programma può essere scritto utilizzando solo tre strutture:

1. Struttura sequenziale.
2. Struttura di selezione.
3. Struttura di iterazione.

Sulla struttura sequenziale non mi soffermerei molto, perché sono le normali istruzioni scritte appunto una di seguito all'altra. Le Strutture di selezione invece consentono, tramite un confronto di "prendere una decisione" e quindi modificando la sequenza, prendendo nuove strade. Infine ci sono le sequenze di iterazione, che detto in soldoni, ma avremo tempo di approfondire, consentono di ripetere (o iterare) una certa sequenza, anche qui ci sarà molto da dire.

4.1 – If/Elif/Else:

If è forse uno degli operatori più utilizzati perché ci consente di eseguire un pezzo di codice o un altro in base ad un confronto, con una variabile, con un numero o altro, ma vediamo subito un esempio:

```
import std/[random, strformat]
randomize()

let casuale = rand(300)
echo(casuale)

if casuale <= 100:
   echo(fmt("Il Numero {casuale} è un Valore Basso"))
   let prova = 1
elif casuale > 200:
   echo("Il Numero ", casuale, " è un Valore Alto")
else:
   echo("Il Numero " & $casuale & "è un Valore Medio")
echo("provo a stampare prova: ", prova)          → Error: undeclared identifier: 'prova'
```

Come puoi vedere l'uso del ***if*** pur nella sua potenza è molto semplice, ma val la pena osservare alcune cose, la prima è che la linea dell'***if*** termina con "***:***" e le istruzioni da eseguire in caso la condizione sia vera (***true***) sono identati di due spazzi dall'if. La seconda che si nota meno ma ce, è l'ambito, ovvero dove una variabile "vive", la variabile *prova* dichiarata dentro al blocco, vale solo ed esclusivamente in quel blocco, aldifuori non esiste. La terza riguarda l'uso di ***if*** ed ***elif***, perché se si fa come in questo caso, il controllo dello stato della stessa variabile, si usa ***if/elif***, cosi facendo il programma alla prima condizione vera, esegue il blocco e non controlla le altre (meglio mettere per prima la condizione più probabile cosi si evita lavoro inutile al processore), mentre se si fanno controlli su variabili diverse ben vengano le sfilze di ***if***. Per quanto riguarda gli operatori di confronto, ormai dovresti conoscerli e sono: <,<=,>=,>,!=. Una menzione speciale va agli opratori ***in*** e ***notin*** che hai gia visto, ma vale la pena rivederli con un esempio perché ti tornerà molto utile:

```
import std/[random]
randomize()
var sequenza: seq[int]
for _ in 1..30:
   var num = rand(200)
   sequenza.add(num)
echo(sequenza)

if 75 in sequenza:
   echo("Il numero 75 è nella sequenza")
else:
   echo("in numero 75 non è nella sequenza")
```

Come puoi vedere con una sola operazione fatta con ***in*** si fa una ricerca in una sequenza (array, stringa...) che può essere anche molto lunga. Una cosa che forse non hai notato è il "_" dopo il ***for*** al posto di una variabile, questo lo si fa quando quel valore non interessa e si dice al compilatore di non tenerne traccia. Per concludere è sempre buona cosa finire con un ***else*** cosi ti assicuri di catturare tutti gli eventi e magari evitare spiacevoli blocchi del programma, ma non è obbligatorio con questo costrutto. Quando usi ***if*** capita spesso di dover fare più confronti simultanei con più variabili, in questo ti vengono in aiuto gli operatori logici come ***or*** e ***and***, tieni presente che dove ce ***and*** entrambe le condizioni devono essere vere, mentre con ***or*** basta sia vera una delle due (o entrambe), ci sono anche altri operatori come ***not*** e ***xor*** che puoi usare, ma ecco un esempio di un programma per la morra cinese che li sfrutta:

```
proc vincitore(a, b: char): int =
   if a == b:
      echo("Pari")
      result = 0
   elif (a == 'c' and b == 's') or (a == 's' and  b == 'f') or (a == 'f' and  b == 'c'):
      echo("Giocatore A Vince")
      result = 1
   else:
      echo("Giocatore B Vince")
      result = 2

echo(vincitore('f', 'c'))                    → Giocatore A Vince (combinazione f = forbice, c = carta)
```

Per una leggibilità migliore ho usate le parentesi ma non son necessarie. Come puoi intuire, basta che una delle tre condizioni ***and*** sia vera, che grazie agli ***or*** tutta la condizione risulterà vera e di conseguenza il vincitore sarà il giocatore "A", altrimenti se nessuna condizione ***and*** risultasse vera l'espressione sarebbe sempre falsa, portando il giocatore "B" alla vittoria.

4.2 – When.

Non so se sia corretto mettere ***when*** in questa sezione, ma essendo molto simile ad *if* mi sembrava la cosa più corretta da fare. Come dicevo ***when*** è molto simile ad *if* perché esegue un blocco di codice solo se è vera la sua condizione, ma anziché operare quando il programma è in esecuzione, interviene in fase di compilazione, facendo ad esempio compilare o meno un pezzo di codice, vediamo un esempio:

```
echo("Stampo in esecuzione...")

when isMainModule:
  echo("questo lo stampo solo se non son un modulo!")

 when system.hostOS == "linux":
   echo("Son un pc Linux")
 when system.hostOS == "windows":
   echo("Son un pc Windows")
 when system.hostOS == "macosx":
  echo("Son un Mac")
```

Qui forse vale la pensa spendere due parole; quando il compilatore trova *when isMainModule* compilerà cio che è identato solo se è il programma principale (main) altrimenti lo salta allegramente. Stessa cosa per esempio con ***when*** *system.hostOS == "linux"* che compilerà solo quella specifica parte di codice se il computer è linux e cosi via.

4.3 – Case/Of:

L'istruzione ***case*** - ***of*** è molto simile ad *if* nel senso che pure essi fanno delle "scelte" ma lavora in modo diverso, ovvero non ce un confronto diretto con una variabile, ma ci son dei "casi" per cui va eseguito un blocco di codice oppure un altro, e questo come vedremo è molto elastico, contrariamente all' *if* che è piu rigido, capirai meglio con un esempio:

```
let confronto = 75

case confronto
of 75:
  echo("Si è 75..")
of 50..80:
  echo("si è sicuramente compreso tra 50 ed 80..")
of 65, 75:
  echo("si è o 65 o 75..")
else:
  echo("Caso non coperto")
```

Puoi facilmente osservare come ***case if*** sia molto elastico; nel primo caso è simile a *if* perché lavora su una variabile, e nel caso questa sia uguale al parametro di confronto, viene eseguito il codice identato, ma gia il secondo caso è interessante, perché grazie all'operatore ".." ***of*** ti consente di fare il confronto con tutti i numeri (continui) in quell'intervallo, nel nostro caso da 50 a 80. anche il quarto caso è interessante perché ti consente di confrontare diversi numeri con la tua variabile (anche discontinui), in questo caso caso 65 e 75, ma ne poi aggiungere quanti ne vuoi. ***Case*** va concluso con l'istruzione ***else*** per catturare tutti quei casi che non ci interessano, ma che comunque potrebbero verificarsi (se ad esempio però usi un *bool* ed usi entrambe le opzioni (*true* e *false*), l'***else*** non occorre perché già copri tutte le opzioni). Da segnalare che ***case/of*** in certe condizioni può essere più performante della controparte *if/else*.

4.4 – While.

Comincerai ora a vedere le strutture di iterazioni di Nim, e si comincia con while, che forse è la più semplice. Questa struttura continuerà ad iterare, indifferentemente da quante volte lo deve fare, fino a che la condizione rimane vera, vediamo un esempio:

```
var x = 1
while x < 5:
   echo("x ora vale: ", x)          → 1,  2, 3 ,4
   inc x
echo("uscito con x = ", x)          → 5
```

L'esempio è molto chiaro, ***while*** viene eseguito fin tanto che "***x***" è miore di 5, poi esce dal ciclo. Cosa molto usata con ***while*** per esempio sui microcontrollori ma non solo, è **while true:** che creerà un ciclo infinito.

4.5 – For.

L'operatore ***for*** ti consente di iterare un numero stabilito di volte, oppure di iterare su un oggetto iterabile, come un array, una sequenza, una stringa, e questo come vedrai utilizzandolo, sarà di un utilità estrema. Qui vedremo più esempi ma molto semplici, perché il ***for*** lo si può usare davvero in moltissimi modi che val la pena analizzare:

```
for x in 1..10:                     → grazie all'operatore ".." for itera da 1 a 10.
   stdout.write(x, "-")             → 1-2-3-4-5-6-7-8-9-10


for x in @[40, 50, 55, 3]:          → for può iterare su una sequenza, un array, un set.
   stdout.write(x, "-")             → 40-50-55-3

for x in "ciao":                    → for puo iterare su una stringa.
   stdout.write(x, "-")             → c-i-a-o

for x in countup(1, 10, 2):         → countup dice a for di iterarare dal primo parametro al secondo (salendo), con il passo
   stdout.write(x, "-")               indicato dal terzo parametro. → 1-3-5-7-9

for x in countdown(20, 10, 2):      → countdown dice a for di iterarare dal primo parametro al secondo (scendendo), con il
   stdout.write(x, "-")               passo indicato dal terzo parametro. → 20-18-16-14-12-10
```

Il ***for*** ha veramente molti caso d'uso e che ti torneranno sempre molto utili, ma il suo uso è davvero molto semplice e non mi dilungherei di più sulla descrizione perché mi sembra molto auto esplicativo.

4.6 – Try/Except.

Gli operatori ***try*** / ***except*** ti consentono di "catturare" un errore in fase di esecuzione evitando cosi la chiusura brutale del tuo programma. Faccio subito un esempio e cerco di chiarirti il concetto:

```
import strutils

let a = stdin.readLine()

try:
  var v = parseInt(a)
  echo v
except ValueError:
  echo ("non converto!!")
```

Questo va spiegato; se non usassimo la struttura ***try***/***except*** e all'input invece di dare un "numero-stringa" tipo "*123*" dai qualcosa tipo "*123x*" il programma andrebbe in errore e terminerebbe anomalamente scrivendo "*Error: unhandled exception: invalid integer: 123x [ValueError]*" ma per fortuna l'eccezione sollevata (in indaco) può essere catturata da ***try*** e gestita da ***except*** in un qualche modo, nel nostro caso scrivendo la non avvenuta conversione. Inutile dirti che questa cosa potrebbe gestire una mancata lettura di un file, una mancata connessione o altro che porti ad un errore fatale. Arriverà un giorno però, in cui tu stesso scriverai delle funzioni che potrebbero provocare degli errori, e sarebbe cosa giusta creare le proprie eccezioni visto e considerato che possono poi tornare assai utili. Queste eccezioni si possono sollevare utilizzando l'operatore ***raise*** e le nuove eccezioni si crea con ***newException*** vediamo un esempio:

```
proc provaEccezione(a, b: int) =
  if a < b:
    raise newException(IOError, "errore a<b")
  else:
    echo("Tutto Bene!")

var a = 3
var b = 7
provaEccezione(a, b)
```

Qui se *a* è minore di *b* solleverà un eccezione che si potrà poi catturare con il ***try*** ed eventualmente intervenire, magari cambiando il valore ad una delle due variabili. Ti propongo qui di seguito, un esempio di come si può gestire un errore e risolverlo con ***try/except***:

```
proc provaEcc(a, b: int) =
  if a < b:
    raise newException(IOError,"errore a < b")
  else:
    echo("Tutto Ok nella funzione!")

var a = 3
var b = 7

try:
  provaEcc(a, b)
except IOError:
  a = 10
  echo("Problema Risolto con except")

provaEcc(a, b)
```

In questo esempio, la procedura *provaEcc()* accetta due valori (a e b) e per qualche ragione se "b" è maggiore di "a", solleva un eccezione, che potrebbe essere un grave errore che manderebbe in crah il programma. Mettendo la chiamata alla funzione dentro ***try***, se viene sollevata un eccezione, non fa chiudere il programma in malomodo, ma come in questo caso ci da la possibilità di gestire ed eventualmente correggere la causa che ha provocato il problema. Sadicamente provoco il problema imponendo "a" minore di "b", ma grazie a questo costrutto, dentro al blocco ***except*** lo puoi correggere imponendo " a = 10" che ora è maggiore di "b" e la procedura può lavorare correttamente.

5 – Le Procedure (Funzioni).

Una ruolo molto importante nella programmazione, è svolto dalle procedure (dette anche funzioni) in Nim in particolar modo perché come linguaggio, pur essendo molto elastico, di suo preferisce il modo procedurale, cioè con uso intensivo appunto di procedure. Ma perché le procedure sono cosi importanti? Principalmente perché consente di suddividere il codice in piccole porzioni che si occupano solo di una cosa, ed è quindi più semplice scriverle e mantenerle e direi a debug-arle (i romani dicevano "divide et impera"), cosi un problema si divide in tanti piccoli sotto problemi molto più facili da risolvere, in più se un giorno ti dovesse servire la stessa procedura per altri programmi, ecco che si può o banalmente copiarla oppure creare un modulo ed importarla nel nuovo programma in maniera semplice e pulita (vedremo in questa sezione come). Un modo per immaginare le procedure, è pensarle come a delle scatole chiuse e che fanno "qualcosa" al loro interno, ma dove per comunicare con esse si può farlo soltanto inviando dei dati nel momento della loro chiamata (*argomenti*) e ricevere il loro risultato alla fine del loro compito (con *result* o *return*).

5.1 – Dichiarazione Di Una procedura.

Il primo passo da fare, è capire come si dichiarano le procedure in Nim, di seguito vedrai alcuni esempi e spero cose utili da sapere:

```
proc nomeProcedura(a: int, b: string): char =
  discard
```

In questa breve dichiarazione già puoi vedere parecchie cose interessanti:

- Un procedura viene dichiarata con la parola chiave ***proc***.
- Il nome deve iniziare con lettera minuscola e non con numeri o caratteri strani.
- Gli argomenti sono scritti tra parentesi e dichiarato il loro *tipo* (oppure a = 3, b = "ciao" ed il compilatore deduce da solo il tipo usato).
- Dopo ":" viene indicato il tipo di valore che la procedura ritorna in questo caso un "char".
- L'operatore "***discard***" in questo caso viene usato come il "pass" di python, ovvero dice al compilatore che la funzione esiste, ma per ora è vuota (senza darebbe errore).
- Dopo il simbolo "=" si scrive il corpo della funzione, che se a capo va identato di due spazzi.

Le procedure, almeno per ora, devono essere scritte prima del corpo principale del programma, altrimenti il nome della funzione non viene trovato, cosi come se la procedura *fx2* dovesse chiamare la procedura *fx1*, la procedura chiamante ovvero *fx2* deve essere dichiarata dopo la funzione chiamata *fx1*. Per evitare problemi che potrebbero presentarsi quando hai molte funzioni, una soluzione è quella di dichiararne i "prototipi", e lo si fa (all'inizio del programma) scrivendo solo la dichiarazione della funzione omettendo "=".

```
proc nomeProcedura(a: int, b: string): char          → senza "=" e posta all'inizio è un prototipo di funzione.
```

Ora la funzione reale puoi davvero metterla dove vuoi, e in qualsiasi ordine, perché verrà sempre ritrovata (cosa che come dicevo semplifica molto la vita quando hai molte funzioni). Un esempio pratico:

```
import sequtils
proc estraiCarattere(stringa: string, indice: int): char      → prototipo della funzione (senza "=").

echo("Ho estratto: ", estraiCarattere("ciao", 2))            → Ho estratto: a
discard estraiCarattere("albero", 3)                         → senza "discard" darebbe errore.

proc estraiCarattere(stringa: string, indice: int): char =   → funzione reale.
  let sequenza = toSeq(stringa)
  result = sequenza[indice]                                  → "result" ritorna un valore ed è auto dichiarato.
```

Anche in questo esempio, puoi vedere altre cose interessanti. Ovviamente la prima è l'uso della dichiarazione del prototipo della funzione all'inizio del programma e come si usa (in verde); ma la cosa più importante è la parola chiave "*result*" che è una variabile riservata, e viene automaticamente associata al tipo dichiarato di ritorno della procedura e va messa come ultima istruzione. Puoi anche usare "*return*" ma l'uso di questo operatore è consigliato solo nel caso in cui la procedura debba ritornare un risultato prima dell'effettiva conclusione della procedura stessa (ad esempio dentro ad un "*while true*"). La funzione potrebbe anche non avere un valore di ritorno, in qual caso non ci vanno i ":" ma solo "=". Altra cosa che puoi notare è "*discard*" nella chiamata della funzione; quando si chiama una funzione che ha un qualsiasi ritorno, ma a te non interessa questo valore, devi dirlo esplicitamente al compilatore cosi non ti creerà problemi sul fatto che lo stai ignorando.

```
proc nomeProcedura() =              → procedura che non ritorna nulla (void).

proc ritorno(): int =               → grazie a return la procedura può uscire dal ciclo while infinito.
  var a: int = 0
  while true:
    if a > 10:
      return a                      → prova a scrivere result = a, e vedi cosa succede!!
    a.inc()

echo(ritorno())
```

La prima procedura non ritorna nulla (quindi non si usa nemmeno *discard*) nella seconda invece, puoi vedere l'uso di *return* in questo caso *result* non funziona, in quanto non è l'ultima istruzione della procedura, quindi va usato *return*.

5.2 – Passaggio di Array e Sequenze come Argomento.

Ad una procedura si può passare anche un intero array o una sequenza, e vedrai in questo paragrafo come si procede, ma faccio subito con un esempio pratico, e passiamo ad una procedura un array:

```
proc sommaAdArray(arry: array[4, int]): array[4, int] =
  var arry_copia = arry
  for indice in arry.low..arry.high:
    arry_copia[indice] = arry[indice] + 10
  result = arry_copia


let array1 = [1, 2, 3, 4]
let array2 = [10, 20, 30, 40, 50]          → troppi elementi da passare alla funzione!!!

echo sommaAdArray(array1)                  → [11, 12, 13, 14]
echo sommaAdArray(array2)                  → Error: type mismatch: got <array[0..4, int]>
```

Quando passi un array ad una procedura, **devi** specificare obbligatoriamente quanti elementi contiene e di che tipo, allo stesso modo devi farlo quando intendi passare dalla procedura al chiamante un array. Se come nel caso di *array2* passi cinque elementi anziché i quattro dichiarati, riceverai un errore in fase di compilazione in altre parole una volta stabiliti quanti elementi dovrai passare alla procedura quelli dovranno sempre essere . Nota che per poter fare delle modifiche al array all'interno della funzione, prima abbiamo dovuto copiarlo, ma questo lo vedrai meglio nel paragrafo 5.3. In modo molto simile, si può passare alla procedura una sequenza, ecco un esempio:

```
proc sommaSeq(sec: seq[int]): seq[int] =
  var sec_copia = sec
  for indice in 0..<len(sec):
    sec_copia[indice] = sec[indice] + 10
  result = sec_copia

let sec1 = @[1, 2, 3 ,4]
let sec2 = @[10, 20, 30, 30, 40, 50, 60]

echo somma_seq(sec1)                       → @[11, 12, 13, 14]
echo somma_seq(sec2)                       → @[20, 30, 40, 50, 60, 70]
```

La prima cosa che puoi notare, è che con la sequenza puoi variare a piacere gli elementi passati senza causare errori (in realtà vedrai nel paragrafo 5.4 che puoi farlo anche con gli array) cosa decisamente comoda. La seconda è che con al sequenza nel ciclo *for* ho usato *<len()* perché come gai detto è più idoneo allo scopo di *low* e *high* usato sull'array. Non mi soffermo sulla dichiarazione, perchè mi sembra molto chiara. Piccola variazione, questa volta ho usato il *camel_case* per ricordarti che anche questo stile è ammesso.

5.3 – Passaggio di Argomenti per Valore / Riferimento.

Qui si entra in un discorso un po' più delicato, ovvero come si passano gli argomenti ad una procedura; generalmente lo si può fare in due modi:

- ***Per valore***, e qui se vuoi modificare questo valore **devi** farne una copia all'interno della procedura e solo dopo puoi modificarlo generalmente più lenta (specie su grosse mole di dati) ma che crea meno problemi.
- ***Per riferimento***, dove viene praticamente passato un puntatore al valore, pratica molto più veloce in fase esecutivo ma pericolosa perché quel valore sarà modificato per chiunque lo richieda.

Come di consueto farò un paio di esempi per illustrare le diverse modalità di chiamata:

```
proc perVal(valore: int): int =          → passaggio per valore.
  var copia = valore                     → viene fatta una copia del dato da modificare.
  copia = copia * 100                    → "copia" ora può essere modificata a piacere.
  result = copia

proc perRef(valore: var int): int =      → passaggio per riferimento.
  valore = valore * 100                  → nessuna copia fatta, ma modificabile direttamente.
  result = valore

var vA = 2
var vB = 4

echo ("vA= ",vA, "vA fx= ", perVal(vA),"vA vale ora= ", vA)   → vA= 2  fx= 200  vA vale ora= 2
echo ("vB= ",vA, "vB fx= ", perRef(vB),"vB vale ora= ", vB)   → vB= 2   fx= 400  vB vale ora= 400
```

Nell'esempio qui sopra, risulta molto evidente quanto detto finora, infatti la procedura *perVal* non va a modificare la valore *vA*, mentre la procedura *perRif* va a modificare il dato originario, cambiandolo per sempre; ho evidenziato la caratteristica che rende una procedura per "valore" e l'altra per "riferimento", ovvero l'aggiunta del operatore ***var*** subito prima della dichiarazione del tipo dell'argomento, in pratica gli dice che "*valore* è un puntatore a *int*", puntatore sicuro tracciato dal GC.

5.4 – OpenArray & Varargs.

Due metodi alternativi per passare argomenti ad una funzione, e meritano un capitoletto tutto loro, sono gli *openarray* e le *varargs*. A volte, non possiamo sapere a priori quanti elementi contiene un array e come hai visto, se vuoi passarlo ad una procedura, devi dichiarare esplicitamente quanti ne contiene e di che *tipo*. Con gli *openarray* risolviamo questo problema (immagino ci sarà un costo in prestazioni da pagare) infatti basta solo dichiarare di che *tipo* sono senza specificarne la grandezza, quindi potremo passare di volta in volta array, (o anche sequenze se dovesse occorrere!), di capacita diverse:

```
proc contaElementi(arr: openArray[int]): int =
  for t in arr:
    result.inc()

let array1 = [1, 2, 3, 4]
let array2 = [1, 2, 3 ,4, 5, 6, 7]
let seq1 = @[1, 2]

echo contaElementi(array1)          → 4
echo contaElementi(array2)          → 7
echo contaElementi(seq1)            → 2
```

Nel esempio sopra, passi alla procedura ogni volta un array (o una sequenza) di dimensioni diverse ed il compilatore non avrà nulla da ridire, cosa che non potresti fare con un array "normale". Le *varargs* come in Python, in vece, ti permettono di passare alla tua procedura un numero di argomenti non noti a priori e questi valori, saranno automaticamente inseriti in un array (dentro alla procedura), da dove li puoi facilmente recuperare:

```
proc argomenti(a: string, b: int, c: varargs[int]) =
  echo("Argomento 1: ", a)          → Nim
  echo("Argomento 2: ", b)          → 1
  echo("Argomento 3: ", c)          → [10, 20, 30]

argomenti("Nim", 1, 10,20 ,30)
```

Anche con gli *openarray* e le *varargs* si potranno usare i *generici* come *tipo* e quindi renderli ancora più flessibili, generici che vedrai nel prossimo capitoletto.

5.5 – Ritorno Dati dalle Procedure.

Fino ad ora hai visto come dare in pasto i dati alla procedura, ma spesso, avrai bisogno di recuperare il risultato prodotto per usarlo nel programma chiamante. Hai già visto molti esempi di ritorno di valori, ma ho pensato di aggiungere un capitoletto apposito per cercare di trattare in modo più specifico e concentrato l'argomento.

```
proc somma(a, b: int): int =
  result = a + b
```

Il valore di ritorno dichiarato in rosso, viene posto dopo gli argomenti di ingresso della funzione, preceduto dai due punti ":" e seguito dal segno "=", questo vuole dire che la procedura in questo caso può solo ritornare valori di *tipo* intero e null'altro, è chiaro che una procedura può ritornare tutti i *tipi* supportati da Nim anche quelli personalizzati che vedremo in seguito (Cap.6) oppure potrebbe non ritornare nella, in quel caso si omette il *tipo* e si mette subito "=" (equivale a void di C). Nim accetta tre strade per restituire il valore:

- ***La prima*** è il valore ricavato dall'ultima riga della funzione sarà inviato al chiamante, in questo caso si poteva scrivere solo : a + b ed il valore di questa somma sarebbe stato automaticamente ritornato.

- ***La seconda*** è l' uso di ***return*** che va usato solo in quei casi dove la riga o il risultato in questione non sia l'ultima della funzione, per esempio dentro un ciclo *while*.
- ***La terza*** è l'uso di ***result*** che deve essere la via preferenziale, da usare il più possibile. La variabile *result* sarà inizializzato automaticamente al *tipo* dichiarato di ritorno della procedura. Dentro ad un *if* ad esempio usa *result* perchè li finisce il blocco (se ci entra).

Non sempre però sarai in grado di terminare che *tipo* la funzione ritorna, quando avrai queste incertezze, si può usare ***auto***, e sarà il compilatore a decidere quale usare; certo se puoi evitarne l'uso è meglio sopratutto per la leggibilità del codice ecco un esempio:

```
proc secondoGrado(a, b, c: float): auto =
```

Ora puoi ritornare, *float*, *int*, *sting*, a seconda di come implementerai la tua procedura, ma chi leggerà l'intestazione di questa procedura avrà qualche problema a capire cosa aspettarsi di ritorno. Un sistema interessante per ritornare più valori contemporaneamente anche non omogenei è quello di usare le *tuple* o ancora meglio le *nameTule* (son delle tuple con i nomi assegnati). L'esempio che ora ti propongo di certo potrebbe essere fatto meglio, ma ti da l'occasione di vedere delle cose nuove:

```
import std/[math, strutils]
proc secondoGrado(a, b, c: float): tuple[r1: string, r2: string] =
  let discriminante = (pow(b, 2.0) - (4 * a * c))
  if discriminante < 0:
    echo("Numeri complessi")
    result = ("Numero Complesso","")
  elif discriminante == 0:
    echo("una soluzione reale")
    let sol1 = (-b / (2 * a))
    echo sol1
    result =  (sol1.formatFloat(ffDecimal, 3),"")
  else:
    echo("due soluzioni reali")
    let sol1 = (-b + sqrt(discriminante)) / (2 * a)
    let sol2 = (-b - sqrt(discriminante)) / (2 * a)
    result =  (sol1.formatFloat(ffDecimal, 3), sol2.formatFloat(ffDecimal, 3))

let equaz = secondoGrado(1, -22, 1)
echo(equaz)                          → (r1: "21.954), r2: "0.046")
echo(equaz.r1)                       → 21.954 (stringa)
```

Come in tutte le equazioni di secondo grado che si rispettino, possiamo avere tre tipi di soluzioni: reali e distinte, reali coincidenti, oppure impossibili (se non usi i numeri complessi) quindi il tipo di risposta che la procedura ci ritorna è inevitabilmente diverso ed a priori non sappiamo quali sia, in questo modo usando la tupla e convertendo i numeri float in stringhe, possiamo ritornare al chiamante o due numeri, o uno solo, oppure un avviso che hai ottenuto un numero complesso. In questo caso la tupla è composta da due stringhe ma puoi metterci qualsiasi tipo anche diversi tra loro. L'operatore ***formatFloat*** converte da float a stringa e se usi come argomento ***ffDecimal*** puoi stabilire quante cifre significative usare.

5.6 – Options.

Può capitare a volte, che una procedura, per diversi motivi, non sia in grado di ritornare il *tipo* di dato che le abbiamo imposto e questo potrebbe o dare un errore in compilazione, o peggio, ritornare un dato errato che potrebbe avere serie conseguenze; pensa ad esempio se durante una media un dato inserito male venisse interpretato come "0.0" anziché con il suo reale valore. Come risolvere questo problema e far si che la procedura ritorni qualcosa come "*niente/none*"? La soluzione migliore sono gli *options*. Ecco un esempio:

```
import std/[options, strutils]

proc inputNumero(): Option[float] =
  stdout.write("Inserisci Un Numero: "); let strNum = stdin.readLine()
  try: result = some(parseFloat(strNum))
  except(ValueError): result = none(float)

let risultato = inputNumero()
if isNone(risultato): echo("Hai Sbagliato a Digitare il Numero")
if isSome(risultato): echo("Il Valore Digitato è: ", risultato.get())
```

In questo piccolo programma ho cambiato sensibilmente lo stile scrivendo le istruzioni sulla stessa riga, cosa che puoi fare se ad esempio ci son i ":" oppure se metti ";" alla fine di un istruzione oppure dopo la "," negli argomenti. Ora la procedura ritorna un *option[float]* che può essere facilmente gestito, per controllare se ti è stato ritornato correttamente il *tipo* che ti aspetti oppure un *none,* se ritorna *none* qualcosa è andato storto ma ora non sarà più un problema. Se ritorna il valore corretto, la sua estrazione avviene con la procedura *get()*. Come di consueto ecco alcune delle procedure:

- ***get***: ritorna il contenuto dell'*option[]*, se non contiene valori si solleva l'eccezione *UnpackDefect*.
- ***IsNone***: Controlla se l'*option[]* è "none".
- ***IsSome***: Controlla se l'*option[]* contiene un valore valido.
- ***none***: Ritorna un *option[]* senza nessun valore (un "none" in pratica).
- ***some***: Ritorna un *optin[]* con un valore (del *tipo* dichiarato tra le quadre).

Si potrebbe usare anche "Nan" ma poi la sua gestione diventa complicata, quindi ad ora le *option* sono la soluzione migliore da usare. Si può anche passare ad procedura *none*, cosa che a volte potrebbe servire, nel seguente modo:

```
import options

proc passaNone(a: int, b =  none(int)) =
  if isNone(b):
    echo "b  vale ", b
  elif isSome(b):
    echo "b vale ", b.get()

passaNone(8), some(7))                  → b vale 7, come ci si aspettava in qaunto passato l'intero 7.
passaNone(6)                            → b vale none(int), ora b vale correttamente none.
```

nella dichiarazione della procedura basta dichiarare il parametro come *none(tipo)*, e nella chiamante o non si mette nulla (= *none*) oppure quel parametro lo si dichiara come *some(tipo)*.

5.7 – Generici.

Nel capitolo 5.1 ho detto che gli eventuali argomenti di una funzione, devono essere dichiarati esplicitamente (*arg1: int, arg2: string*) oppure implicitamente (*arg1 = 75, arg2 =”Nim”*) ma questo non sempre è possibile. Ci sono casi in cui a priori non è possibile sapere il tipo di dato che invierai alla tua funzione, oppure casi in cui la funzione potrebbe lavorare tranquillamente con tipi simili (ad esempio *int* e *float*) e si vuole evitare inutili e pesanti duplicazioni del codice, ovvero riscrivere praticamente la stessa cosa per i due tipi. Per risolvere questo problema ci vengono in aiuto i generici. Con i generici puoi passare (in teoria) qualsiasi tipo alla nostra funzione, a patto che siano compatibili con ciò che la funzione stessa fa. Ma faccio un esempio:

```
proc somma[T](num1: T, num2: T): T =
  result = num1 + num2

echo(somma(3, 5))                  → 8
echo(somma(3.14 , 10.1))           → 13.24
echo(somma('N', 'i'))              → Errore! 'N' e 'i' non possono essere sommati!
echo(somma(3, 5.4))                → Errore! 3 e 5.4 sono tipi diversi e T non può assumere due tipologie!
```

Un generico lo si dichiara con **[T]** prima degli argomenti stessi, esso assumerà il tipo a seconda di cosa la chiamata manderà alla procedura, ovviamente può assumere un solo tipo alla volta, infatti nel quarto esempio che forziamo a due tipi diversi ritorna errore in compilazione. Nel primo caso **T** sarà di tipo *int* nel secondo caso sarà *float*. Da notare che anche il dato ritornato dalla procedura essendo di tipo **T** dovrà ritornare lo stesso tipo di quelli mandati. Il terzo caso pur avendo gli argomenti dello stesso *tipo*, darà errore in compilazione, perché i caratteri non si possono sommare tra loro. Potremo anche avere l'esigenza di limitare i tipi che un generico può ricevere magari per evitare spiacevoli sorprese in fase di esecuzione, Nim ti permette di fare questo, semplicemente dichiarando la procedura come:

```
proc somma[T: int | float](num1: T, num2: T): T =
```

In questo modo il generico può accettare soltanto numeri *int* oppure *float* tutte le altre cose solleveranno un errore (in compilazione). E se avessi più argomenti che però potrebbero (o no!) essere di tipo diverso, come si potrebbe fare, vediamo un esempio:

```
proc cerca[T, S](chiave: T, lista: S): bool =
  if chiave in lista:
    result = true
  else:
    result = false

echo(cerca('f', ['b' ,'j', 'a', 'z']))   → false
echo(cerca(3, [1, 5, 3, 9, 0]))          → true
echo(cerca(1.5, [1.0, 5.7, 1.5, 9.3]))   → true
```

In questo semplice esempio, passi alla procedura due argomenti, il primo fa da chiave, il secondo come un elenco di varie cose. In questo caso due generici, **T** varrà *char* o *int* o *float*, mentre **S** sarà di tipo *array[char]* o *array[int*] oppure *array[float]* ecco dunque che alla procedura ora si passano due *tipi* diversi.

5.8 – Proc vs Func.

In Nim ci son due modi per dichiarare un procedura, ovvero **proc** e **func**, ma in cosa differiscono le due modalità? La differenza è sottile, ma importante. Come prima approssimazione, puoi immaginare questa differenza come quella tra *var* (***proc***) e *let* (***func***), func è molto più restrittiva rispetto a proc, questo vuole dire che func non può arrecare nessun tipo di modifica a qualcosa esterno a se stessa (gli argomenti passati son interni!!), mentre *proc* lo può serenamente fare (grazie per i chiarimenti Pietro Peterlongo). Due esempi classici di modifiche esterne sono:

- Stampare a video con "echo" questo semplice comando viene interpretato come una modifica al terminale, quindi una modifica esterna alla procedura ***func***.
- La modifica/lettura di una variabile globale, questa variabile essendo dichiarata al di fuori di ***func*** non può essere modificata ne usata internamente.

Ecco un esempio con i relativi risultati:

```
let globale = 20

proc somma(a: int): int =
  result = globale + a

func diff(a: int): int =
  result = globale - a

echo(somma(10))          → 30
echo(diff(10))           → Errore! 'diff' can have side effects
```

Mentre in *somma()* che usa una variabile globale (è sempre sconsigliato l'uso di variabili globali!!) è ammesso, in ***func()*** non lo è e questo vale per qualsiasi cosa che ti venga in mente che vada ad avere effetti fuori di essa. In pratica ***func*** è un alias per ***proc() {.noSideEffect.}***. In altre parole ***func*** garantisce che non si siano effetti indesiderati come modifiche involontarie nel tuo codice.

5.9 – Proc vs Method.

In Nim ce un altro modo ancora per dichiarare un funzione, ovvero ***method*** che va un po' oltre allo scopo di questo manuale, ma lo menziono perché tornerà utile nella programmazione ad oggetti. Per ora ci basti sapere che ***proc*** è una procedura statica, mentre ***method*** è dinamica, ricordatelo quando vedrai la programmazione ad oggetti, in modo particolare quando ci saranno di mezzo le "ereditarietà" che a volte potrebbero dare problemi se si usasse ***proc***. Questo vantaggio però, lo si paga in prestazioni, perché il ritorno di ***methd*** sarà valutato in fase di esecuzione, invece che, come nel caso di ***proc*** in fase di compilazione. La dichiarazione è praticamente uguale a quanto sai già:

```
method (argomento: int): int =
```

5.10 – Procedure Pubbliche e Private.

Il concetto di procedure pubbliche e private, non è molto importante quando si lavora su di un singolo file, ma diventa cruciale quando si creano librerie che poi vengono importate in un' altro progetto specie se si usa un approccio "ad oggetti" (capitolo 6), perché alcune procedure potrebbero essere "pericolose" da usare liberamente o perchè richiedono prima che delle altre procedure impostiamo dei valori. Qualsiasi procedura in Nim di default è privata (non visibile al di fuori del file di dove è dichiarata), per renderla pubblica bisogna dirlo esplicitamente. Rendere pubblica una funzione è estremamente facile, basta mettere **"*"** dopo il nome della procedura stessa. Guarda come:

```
proc privata() =                    → Questa funzione NON sarà visibile (importato il file che la contiene).
  discard

proc pubblica*() =                  → Questa funzione sarà visibile (importato il file che la contiene).
  discard
```

Questo concetto che per ora può sembrare un po' strano, nel prossimo capitolo sarà spero più chiaro e potrai capire perché è cosi importante la distinzione tra pubblico e privato.

6.0 – Programmazione ad Oggetti.

Ormai sai che Nim preferisce una programmazione procedurale, ma questo non ci impedisce di usare un' altro stile molto usato, ovvero la programmazione ad aggetti che a volte può essere molto utile. In altri linguaggi come per esempio Python, ce una parola chiave riservata che ci permette di creare una classe, ovvero *class*, in Nim non esiste nulla del genere ma in qualche maniera la si può simulare, in questo capitolo vedrai comunque come risolvere questo "problema" e creare qualcosa di molto simile ad un oggetto. Per fare questo, per prima cosa dovrai studiare ***type*** che finora lo abbiamo solo intravisto, ma senza soffermarci troppo su di esso al capitolo 3.

6.1 – Type.

Siediti comodo, perché questo operatore *type* dall'aria cosi mansueta, in realtà ci riserverà parecchie sorprese. Lo scopo principale di *type* è quello di creare delle strutture dati personalizzate dall'utente, in altre parole oltre ai tipi standard come *int*, *char*, *ecc* possiamo creare dei *tipi* che contengo combinazione dei precedenti e molto, molto altro. Cominciamo subito con un esempio perché ci sarà già molto da dire:

```
type
  Frutta = object
    nome: string
    colore: string
    peso: int

var frutto1 = Frutta(nome: "Arancia", colore: "arancione", peso: 100)
echo frutto1.nome                    → Arancia
echo frutto1                         → Visualizza tutto il contenuto della variabile.
                                       (nome: "Arancia", colore: "arancione", peso: 100)
```

Nella prima riga, ce *type* e di seguito identato (come di consueto ho usato 4 spazi anzichè 2 per maggior visibilità) il nome di questo nuovo *tipo* (Frutta), se osservi, inizia con una lettere maiuscola, questo perché in Nim per convenzione, i *tipi* definiti dall'utente e solo essi dovrebbero iniziare in questo modo. Dopo il nome possiamo osservare un'altra istruzione, ovvero *object* e qui mi tocca aprire una parentesi, perché i *type* possono essere di tre tipologie:

1. Gli ***object***: sono grosso modo le strutture di C, dove all'interno ci sono tutte le variabili o campi richiesti. Nel caso degli ***object*** due *type* che son strutturalmente uguali, cioè che contengono gli stessi campi, con gli stessi nomi sono entità **diverse** (vedi esempio sotto).
2. Le ***tuple***: assomigliano molto agli *object*, anche esse son delle strutture dati. Nel caso delle ***tuple*** due *type* che son strutturalmente uguali, cioè che contengono gli stessi campi, con gli stessi nomi sono entità **uguali** (vedi esempio sotto).
3. Gli ***enum***: è simile alle numerazioni di C. definisce una raccolta di identificatori, a cui è associato significato.

```
type
  AgenteCiaObj = object
    nome: string
    eta: int
  AgenteMi6Obj = object
    nome: string
    eta: int

let agente1 = AgenteCiaObj(nome: "Nim", eta: 33)
let agente2 = AgenteMi6Obj(nome: "Nim", eta: 33)

  echo(agente1 == agente2)          → qui riceverai un errore in compilazione, perché AgenteCiaObg ed
                                      AganteMi6Obj, son cose diverse per avendo gli stessi campi con gli
                                      stessi valori.

type
  AgenteCia = tuple
    nome: string
    eta: int
  AgenteMi6 = tuple
    nome: string
    eta: int

let agente3: AgenteCia = (nome: "Nim", eta: 33)
let agente4: AgenteMi6 = (nome: "Nim", eta: 33)

echo(agente3 == agente4)            → qui la risposta è "true" e non genera errori, perchè la struttura
                                      interna ed i campi hanno lo stesso medesimo valore!!
```

Le restanti tre linee vanno a definire il "contenuto" del nostro nuovo *tipo* e vanno dichiarati semplicemente specificando a loro volta di che *tipo* sono. Una volta descritta la struttura che volgiamo usare, va sempre posta all'inizio o per lo meno prima di usare il nuovo tipo come per le procedure, andrai ad usarla come fosse un qualsiasi *tipo* di sistema (come nell'esempio) ora qui potrai inizializzare tutti o solo alcuni dei suoi valori. Nell'ultima linea ti mostro come richiamare un solo valore della variabile **frutto** in questo caso **nome** che è uguale ad "Arancia", puoi anche vedere tutto il contenuto dei campi semplicemente usando solo il suo nome, e si può modificarne il contenuto scrivendo:

```
frutto1.nome = "Mandarino”
frutto1 = Frutta(nome: "Pesca", colore: "Giallo", peso = 123)
echo(frutto1)                          → (nome: “Pesca”, colore: “Giallo”, peso: 123)
```

Non sempre però potrai modificare i valori di questo *tipo* (e qui ci avviciniamo alla programmazione ad oggetti) e questo avviene quando dovrai modificare una o più voci di esso, tramite una procedura, questo perché se ti ricordi dal capitolo precedente, normalmente gli argomenti vengono passati per valore quindi lei potrebbe agire al più su una copia del valore ma non sul valore originale. Ma ce una via d’uscita, usare ***ref*** nella definizione del tuo *tipo*. Ecco subito un esempio che ti chiarirà il concetto:

```
type
  PerValore = object                   → I tipo verrà passato per valore, quindi il compilatore darà errore perchè la
    valore: int                          procedura non potrà andare  a variare quanto passato.

proc cambio(a: PerValore) =
  a.valore =  100                      → errore!! non può essere variato!!
  echo(a.valore)

var x = PerValore(valore: 50)
cambio(x)
echo(x.valore)
```

Se provi a compilare questo codice, il compilatore darà errore dicendo che ***a.valore*** non può essere riassegnata in questo modo, ma se riscrivi il programma nel modo proposto qui sotto le cose cambieranno:

```
type
  PerRiferimentoObj = ref object       → Ora si che può variare il valore passato, in quanto passato per riferimento!
    valore: int

proc cambio(a: PerRiferimentoObj) =
  a.valore =  100
  echo(a.valore)                       → 100

var x = PerRiferimentoObj(valore: 50)
cambio(x)
echo(x.valore)                         → 100
```

Con questa piccola modifica, ovvero aggiungendo ***ref*** prima di *object* il tuo nuovo tipo sarà sempre passato per riferimento (un puntatore tracciato e sicuro) e quindi sempre modificabile dalle procedure.

6.2 – Programmazione ad Oggetti.

Ci siamo, adesso hai tutto ciò che ti può serve per scrivere un programmino in stile ad oggetti. Come accennavo prima, in Nim non esiste la parola chiave *class* che raggruppa al suo interno tutti i metodi che servono per gestire la classe stessa, però possiamo creare un file che come vedrai farà qualcosa di simile ad una classe, e una volta importato (in un secondo programma) sembrerà di lavorare con una vera e propria classe. In questo *import*, *type* e *, ci saranno davvero molto utili. Il prossimo esempio sarà il primo file dove scriveremo la nostra "classe" che poi importeremo, ecco come fare (chiamalo ***persone.nim***):

persone.nim:

```
type
  PersonaObj* = ref object
    nome: string
    anni: int

proc newPersonaObj*(nome: string, anni: int): Persona =
  result = Persona(nome: nome, anni: anni)
proc cambiaNome*(self: Persona, nome: string) =
  self.nome = nome
proc cambiaAnni*(self: Persona, anni: int) =
  self.anni = anni
proc stampa*(self: Persona) =
  echo(self.nome," Ha ", self.anni," anni")
```

Le cose già viste prima, ma importanti, le ho scritte in blù ovvero *ref* e "*". ***ref*** fa si che il *tipo Persona* venga passato per riferimento (quindi modificabile dentro le procedure), ***** fa si che le cose marchiate con esso siano visibili al di fuori del file ***persone.nim***. La novità, scritta in rosso, è ***self*** a cui si assegna il *tipo Persona*. Self non è una parola riservata, quindi si potrebbe mettere qualsiasi cosa, ma ho rispettato la convenzione di Python, ad ogni modo sta a significare che il primo parametro accettato dalla procedura è di tipo *Persona* dove saranno memorizzati i dati e potrai modificarli. Nella programmazione ad oggetti, per inizializzare, cambiare o altre operazioni sull'oggetto stesso, devono sempre essere effettuate tramite le apposite procedure che si occupano di prevenire o correggere eventuali errori che potrebbero mandare in errore il programma e non con l'accesso/modifica diretta dei parametri. Nulla ci vieta ovviamente di usare quanto sopra direttamente magari con l'uso di "*when isMainModule*" ma in questo caso avremo accesso a tutto senza distinzione tra pubblico e privato "nome" e "anni" non saranna acessibili dall'esterno in quanto sono privati!). Ora crea un altro file (che chiamerai ad esempio ***organico.nim***) e scrivi:

Quando istanzi un oggetto, lo puoi fare in due modi, o come vedrai nel prossimo esempio e che mi sento di consigliartelo sempre, ovvero creando una procedura che lo faccia, e questa va fatta quando il file è importato e non ce accesso ai campi del tipo! Mentre se sei nello stesso file della classe puoi anche fare:

```
let uomo = PersonaObj(nome: "Nim", anni: 33)
```

Che poi è vero che fa esattamente la stessa cosa che fa la procedura ***newPersonaObj()*** ma questo ti consente di avere sempre accesso a quei campi anche se un domani questa classe la importerai, ma avrai già pronto all'uso il suo bel costruttore.

organico.nim

```
import persone

let uomo1 = newPersona(nome = "Nimroid", anni = 45)
let donna1 = newPersona(nome = "Athena", anni = 23)

stampa(uomo1)                    → Nimroid Ha 45 anni
stampa(donna1)                   → Athena Ha 23 anni

uomo1.cambianome("Nim")
donna1.cambiaAnni(20)

stampa(uomo1)                    → Nim Ha 45 anni
stampa(donna1)                   → Athena Ha 20 anni
```

In questo semplice programma abbiamo istanziato due oggetti (*uomo1* e *donna1*) con una procedura di inizializzazione (*newPersona()*) ma la cosa più importante è che ogni operazione, fin anche la stampa, ha una sua specifica procedura. Prova ora ad esempio nel file ***persone.nim*** a levare "*" nella procedura *stampa()* oppure nel file ***organico.nim*** a scrivere "*echo("uomo1.nome)*" e vedere cosa accade.. non funzionerà perché entrambi ora sono privati e quindi non accessibili! Ovviamente nessuno ci vieta di scrivere la nostra "classe" nello stesso file dove risiede il programma principale, cosi però il concetto di privato e pubblico perde di significato, perché tutte le procedure ed il resto sarà accessibile.

6.3 – Ereditarietà.

Nella programmazione ad oggetti non può mancare il concetto di ereditarietà, che in parole molto povere è quando una classe eredita procedure, variabili, ecc da una classe superiore (classe radice o madre). Questo è un bel vantaggio, perche ti permette di prendere una classe già esistente e non solo sfruttare quello che ha da offrirci, ma di aggiungere cose specifiche di cui hai bisogno, ma anche qui ti propongo un esempio che trovo molto lampante (per semplicità faremo tutto in un unico file, quindi senza procedure private e pubbliche):

```
type
  CiboRef = ref object of RootObj
    proteine: int
    carboidrati: int
    grassi: int

proc newCibo(prot, carb, garss: int): Cibo =
  CiboRef(proteine: prot, carboidrati: carb, grassi: garss)

proc calcolacalorie(self: Cibo): int =
  result = (self.proteine * 4 + self.carboidrati * 4 + self.grassi * 9)

type
  VerduraRef = ref object of Cibo

proc newVerdura(prot, carb: int, grass = 0): Verdura =
  VerduraRef(proteine: prot, carboidrati: carb, grassi: grass)

when isMainModule:

  var pasta = newCibo(prot = 12, carb = 72, garss = 1)
  echo pasta.calcolacalorie                    → 345

  var melanzana = newVerdura(prot = 2, carb = 3)
  echo melanzana.calcolacalorie                → 20
```

Si lo so, i due tipi *Cibo* e *Verdura* andrebbero scritti di seguito sotto allo stesso *type*, ma cosi facendo ho voluto evidenziare le due "classi" con le proprie procedure. I diversi colori stanno ad indicare a quale delle due classi appartengono le procedure oppure quale chiama il costruttore. Ma veniamo alle cose importanti (quelle in rosso), la prima cosa è ***of RootObj*** che indica che è l'oggetto base da cui poi gli altri oggetti erediteranno, poi chi eredita, nel nostro caso *Verdura* sarà indicato come ***of Cibo***, che come si può intuire, eredita da *Cibo*. In questo semplice caso, le due classi si differenziano per il fatto che una (*Verdura*) ha sempre grasso = 0 mentre in cibo più generica lo si può variare. Infine la procedura ***calcolacalorie()*** la si può usare anche con *verdura,* in quanto come già detto, viene ereditata dalla classe madre. Vediamo ora un caso particolare, hai visto nel capitolo 5.6 l'esistenza di ***method*** e sai che lo si deve usare solo dove non ci son alternative, in quanto è poco performante, ma nel seguente caso sarà obbligatorio usarlo (grazie Andrea Manzoni):

```
type
  Frutto = ref object of RootObj
    nome*: string
  Mela = ref object of Frutto
  Pera = ref object of Frutto

method eat(f: Frutto) {.base.} =
  echo "Sono un Frutto Generico"

method eat(f: Mela) =
  echo "Sono un Mela"

method eat(f: Pera) =
  echo "Sono un Pera"

let cesto = @[Mela(nome:"a"), Pera(nome:"b")]

for frutto in cesto:
  frutto.eat()                    → Sono una Mela  Sono una Pera
```

In questo caso, se provi a sostituire *method* con *proc* vedrai che il programma non funziona, perché la variabile nel *for* cioè cesto, è come se diventasse dinamicamente (in fase quindi di esecuzione) un tipo ogni volta diverso, questo *proc* non lo accetterebbe in quanto il suo tipo viene fissato in fase di compilazione. Ripeto, *method* è da evitare ed usare solo dove non se ne può fare a meno, più comodo si, ma meno performante.

7.0 – Uno Sguardo alle Librerie di Base.

Questo capitolo vuole essere una rapida carrellata su alcuni moduli e procedure di Nim, che mi sembrano molto utili e da tenere sempre sotto mano in caso di dubbi o difficoltà, o comunque un promemoria rapido per vedere se esiste quello che ci potrebbe servire o meno, anche qui si vedranno solo cose per operare a livello base, ci saranno sicuramente alcune ripetizioni di quanto hai visto finora, ma anche cose nuove. Ovviamente ti rimando a https://nim-lang.org/1.6.0/lib.html per la lista completa, perché qui per ovvi motivi la lista dei metodi non può essere completa.

Importazione dei moduli:

```
import std/[math, random]          → con import unisci al tuo spazio nuove definizioni.
from math import cos, log          → con from importi dal modulo indicato solo la/e funzione/i indicate.
```

std/strutils:

```
count(s: string; sub: char): int                          → conta quante volte il carattere/stringa "b" compare nella stringa "a".
countLines(s: string): int                                → conta quante linee ci sono in "s".
delete(s: var string; first, last: int)                   → cancella dalla stringa i caratteri indicati dallo slice.
fromHex[T: SomeInteger](s: string): T                     → trasforma una stringa esadecimale in un intero (da indicare).
intToStr(x: int; minchars: Positive = 1): string          → converte un intero in stringa.
join(a: openArray[string]; sep: string = ""): string      → unisce gli elementi del contenitore separati da un separatore indicato.
parseFloat(s: string): float {.....}                      → trasforma una stringa in un numero float.
```

parseInt(s: string): int → trasforma una stringa in un numero intero.

repeat(c: char; count: Natural): string → ripete "x" volte il carattere indicato.

split(s: string; sep: char;
maxsplit: int = -1): seq[string] → separa la stringa indicata dove ce il carattere indicato e mette in un contenitore.

strip(s: string; leading = true; trailing = true;
chars: set[char] = Whitespace): string → rimuove gli spazi all'inizio o alla fine o entrambe in una stringa.

trimZeros(x: var string; decimalSep = '.') → elimina gli zeri finali (in virgola mobile)

std/algorithm:

isSorted[T](a: openArray[T];
order = SortOrder.Ascending): bool → controlla se un contenitore è ordinato oppure no.

merge[T](result: var seq[T]; x, y: openArray[T]) → unisce due contenitori (presunti ordinati).

reverse[T](a: var openArray[T]) → inverte (gira) il contenuto del contenitore passato.

reversed[T](a: openArray[T]): seq[T] {.inline.} → inverte (gira) il contenuto del contenitore passato e ritorna una nuova sequenza.

sort[T](a: var openArray[T]; order = SortOrder.Ascending) → ordina il contenitore in modo Descending o Ascending.

sorted[T](a: openArray[T]; order = SortOrder.Ascending): seq[T] → ordina il contenitore in modo Descending o Ascending e ritorna una nuova sequenza.

std/sequtils:

count[T](s: openArray[T]; x: T): int → ritorna il numero di occorrenze di "x" nel contenitore "s".

delete[T](s: var seq[T]; slice: Slice[int]) → cancella tutti gli elementi indicati nello slice (se fuori indici solleva RangeDefect).

insert[T](dest: var seq[T]; src: openArray[T]; pos = 0) → inserisce gli elementi di "scr" in in "dest" dalla posizione "pos".

map[T, S](s: openArray[T]; op: proc (x: T): S {.closure.}): seq[S] → ritorna una sequenza con il risultato ottenuto dalla procedura "op" applicato ad ogni elemento di "s".

maxIndex[T](s: openArray[T]): int → ritorna la posizione del elemento più grande nel contenitore.

minIndex[T](s: openArray[T]): int → ritorna la posizione del elemento più piccolo nel contenitore.

repeat[T](x: T; n: Natural): seq[T] → ritorna una sequenza dove "x" è ripetuto per "n" volte.

unzip[S, T](s: openArray[(S, T)]): (seq[S], seq[T]) → ritorna due sequenze dove scompatta i due campi di una tupla a due elementi: @[(1, 'a'), (2, 'b')] → @[1, 2] e @['a', 'b']

zip[S, T](s1: openArray[S]; s2: openArray[T]): seq[(S, T)] → ritorna una nuova sequenza di tuple accostando elemento per elemento il contenuto di due sequenza (contrario un unzip): @[6, 7] e @['v', 'j'] → @[(6, 'v'), (7, 'j')].

toSeq(iter: untyped): untyped → trasforma ogni cosa iterabile in una sequenza.

mapIt(s: typed; op: untyped): untyped → torna una nova sequenza dove i valori ottenuti da "op" son ricavati dal conteiner "s" di ingresso (è una macro.)

std/random:

initRand(seed: int64): Rand → inizializza random per fornire numeri casuali con un seme indicato se non indicato viene posto uguale a 0.

randomize() → inizializza una nuova sequenza di numeri casuali.

shuffle[T](r: var Rand; x: var openArray[T]) → mescola gli elementi di un array in modo casuale.

rand(max: float): float → ritorna un numero casual da 0 a "max" il tipo tornato, dipende dal tipo dato come argomento (,in questo caso float, ma potrebbe essere int.

std/math:

cos(x: float32): float32 → calcola il coseno di “x”.
exp(x: float32): float32 → calcola l’esponente di “x”.
ln(x: float32): float32 → calcola il logaritmo naturale di “x”.
log[T: SomeFloat](x, base: T): T → calcola il logaritmo in base “base” di “x”.
log2(x: float32): float32 → calcola il logaritmo in base 2 di “x”.
log10(x: float32): float32 → calcola il logaritmo in base 10 di “x”.
`mod`(x, y: float64): float64 → calcola il modulo tra due valori float (float 32 o float 64).
pow(x, y: float32): float32 → calcola la potenza tra due numeri float32 o 64 (^ per gli interi).
round[T: float32 | float64](x: T; places: int): T → arrotonda il numero float “x” con “places” decimali (per ora usa ---flag)
sin(x: float32): float32 → calcola il seno di “x”.
sqrt(x: float32): float32 → calcola la radice quadrata di “x”.
sum[T](x: openArray[T]): T → calcola la somma degli elementi di un array.
tan(x: float32): float32 → calcola la tangente di “x”.
trunc(x: float64): float64 → tronca brutalmente i decimali di “x”.

std/tables:

add[A, B](t: TableRef[A, B]; key: A; val: sink B) → aggiunge alla tabella un nuovo elemento (chiave-valore).
clear[A, B](t: OrderedTableRef[A, B]) → cancella tutta la tabella dai suoi elementi.
contains[A, B](t: OrderedTable[A, B]; key: A): bool → se la tabella contiene e la chiave indicata ritorna “true” alias haskey.
del[A, B](t: OrderedTableRef[A, B]; key: A) → cancella la chiave passata con il suo valore.
getOrDefault[A, B](t: OrderedTable[A, B]; key: A): B → recupera il valore della chiave indicata, se non la trova ritorna B.
hasKey[A, B](t: OrderedTable[A, B]; key: A): bool → ritorna “tue” se la chiave indicata è nella tabella.
hasKeyOrPut[A, B](t: var OrderedTable[A, B]; key: A; val: B): bool → ritorna “tue” se la chiave è in tabella altrimenti lo inserisce.
inc[A](t: CountTableRef[A]; key: A; val = 1) → incrementa il valore (di uno di default) indicato dalla chiave passata.
len[A, B](t: OrderedTable[A, B]): int → ritorna il numero di elementi (chiave-valore) contenuti nella tabella.
mgetOrPut[A, B](t: OrderedTableRef[A, B]; key: A; val: B): var B → recupera il valore della chiave se non presente lo aggiunge.

Indice Alfabetico