Raku (autore: Vittorio Albertoni)

Premessa

Agli albori di Internet, esattamente nel gennaio del 1988, quando ancora la rete collegava soltanto alcune decine di migliaia di postazioni, per lo più localizzate in università e centri di ricerca, ad opera di Larry Wall nacque il linguaggio di programmazione Perl.

Nemmeno c'era ancora il World Wide Web come lo conosciamo oggi, inventato presso il CERN tre anni dopo e nemmeno c'erano i browser grafici che usiamo oggi, il primo dei quali, Mosaic, è apparso nel 1993.

Le prime applicazioni del linguaggio Perl, rigorosamente ristrette al mondo Unix che governava le postazioni di lavoro dei precursori di Internet, riguardarono l'esame di file di testo e l'estrazione di informazioni da questi file al fine di produrre report basati sulle informazioni stesse. Il nome del linguaggio è infatti l'acronimo di Practical Extraction and Report Language.

L'ambiente di utilizzo del linguaggio era il più adatto allo sviluppo collaborativo e, dalle origini, Perl è open source.

In pochi anni si arricchì e divenne uno dei primi linguaggi utilizzati per il World Wide Web: la sua affermazione pubblica avviene nel 1991, con l'uscita del testo «Programming Perl» ad opera di Larry Wall e Randal Schwartz.

Da allora è stato un crescendo di arricchimenti, fino alla versione 5 del linguaggio, rilasciata nel 1994 con un interprete completamente riscritto e, a giugno 2020, si è arrivati alla versione 5.32.0.

Le caratteristiche del linguaggio, che è un linguaggio di scripting, sono la semplicità della sintassi, l'eleganza e il dinamismo.

Fin dal 2000, mentre si lavorava a continui aggiornamenti della versione 5, che tuttora continuano, si pensò ad una rifondazione del linguaggio, all'insegna del concetto che "cose semplici debbono restare semplici, cose difficili debbono diventare più semplici e cose impossibili dovrebbero diventare difficili".

La comunità che lavorò a questo progetto di rifondazione ne fece il regalo di Natale e di compleanno per Larry Wall rilasciando la versione 6 di Perl il 25 dicembre 2015.

Perl 6 è ancora più bello di Perl 5. E' di nuova concezione, pur conservando qualche compatibilità con Perl 5. Ma la più grossa novità è il nuovo compilatore di bytecode Rakudo che fa girare gli script Perl sulla velocissima Moar Virtual Machine. Da questa radicale innovazione deriva il nuovo nome definitivamente sancito per Perl 6 nell'ottobre 2019: Raku[1].

In questo manualetto, seguendo uno schema illustrativo che ho già usato per presentare le basi di altri linguaggi di programmazione, penso di mettere in grado il principiante di arrivare a divertirsi creando qualche semplice programma.

Data la caratteristica di Raku di consentire di fare le stesse cose in modi diversi, cercherò di dimostrarlo.

Per approfondimenti segnalo l'indirizzo *https://raku.guide/it/* dove si trova una guida in lingua italiana.

La documentazione ufficiale, in lingua inglese, si trova all'indirizzo *https://docs.raku.org/*.

In entrambi i casi un po' di chiarezza in più non avrebbe guastato: si tratta, infatti, di documentazione non di semplice consultazione per un principiante, soprattutto se si voglia sapere come si fa esattamente una certa cosa.

[1] Visto che il marchio/mascotte di Raku è una strana figura di farfalla detta Camelia, così nominata forse in omaggio al Camel che è il marchio/mascotte di perl 5, è mia opinione che il nome Raku derivi da quello dell'artista giapponese Raku Inoue, la cui specialità è quella di creare modellini di insetto, tra cui farfalle, attraverso collage di fiori, foglie e loro frammenti.

Indice

1 Installazione

Raku si trova all'indirizzo *https://raku.org/*, dove, per prima cosa, ci viene presentata Camelia, la farfalla marchio/mascotte che Larry Wall ha scelto per Perl 6

Sfogliando le pagine del sito possiamo trovare cose interessanti su Perl 6.

Ciò che qui interessa è la pagina DOWNLOAD, nella quale veniamo informati che per lavorare con Perl 6 occorre implementare il compilatore Rakudo e ci viene fornito il link che ci porta all'indirizzo *https://rakudo.org/*.

Dalla home page di questo sito, per una installazione completa di tutto ciò che serve (compresi alcuni moduli base in arricchimento del linguaggio), dobbiamo scegliere la pagina STAR BUNDLE e, in questa, cliccare sul pulsante previsto per il nostro sistema operativo e seguire le istruzioni che ci vengono date.

Alternativamente possiamo andare all'indirizzo *https://raku.guide/it/* e seguire le istruzioni che troviamo nel paragrafo 1.3 COME INSTALLARE PERL 6.

Chi usa Linux molto probabilmente (sicuramente se usa Ubuntu) può installare il tutto affidandosi al gestore dei programmi della sua distro, verificando che vengano installati i pacchetti `moarvm`, `nqp`, `perl6` e `rakudo`. Ovviamente si deve trattare di un rilascio Linux successivo al 2015, meglio ancora se successivo al 2018 per avere una versione abbastanza aggiornata.

Nel momento in cui scrivo (maggio 2021) la versione più aggiornata è la 2021.04, rilasciata il 24 aprile 2021 per Linux; un po' meno aggiornate le versioni per Windows e Mac OS.

2 Come funziona

Raku è una via di mezzo tra i linguaggi interpretati e quelli compilati. Il compilatore Rakudo, infatti, compila il nostro programma non proprio in linguaggio macchina, come avviene per i linguaggi compilati come C, C++, Pascal, ecc., ma in una via di mezzo, in bytecode, come avviene per il linguaggio Java, e gira su una macchina virtuale (la Moar Virtual Machine) compresa in ciò che abbiamo installato.

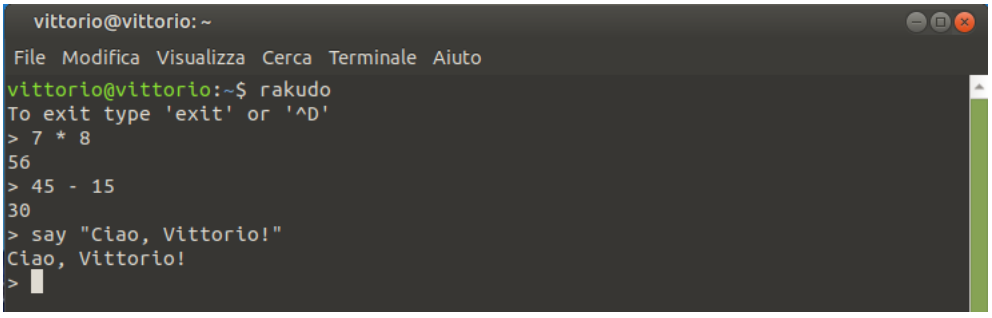
Possiamo utilizzare Rakudo interattivamente o per eseguire programmi salvati su file.

L'utilizzo interattivo ci consente di verificare istantaneamente ciò che facciamo ed è molto utile per imparare il linguaggio.

Se passiamo a terminale (prompt dei comandi per chi lavora su Windows) il comando

`rakudo`

il terminale diventa un terminale Raku



e vi possiamo digitare istruzioni nel linguaggio Raku vedendole immediatamente eseguite.
Questo terminale modificato si chiama REPL (Read Eval Print Loop).
Nella figura abbiamo due semplici operazioni aritmetiche e un saluto.

Possiamo mettere più istruzioni sulla stessa riga separandole con punto e virgola (`;`).
Per istruzioni complesse, come le subroutine, possiamo andare a capo fino a quando sono completate e solo dopo completamento potranno essere eseguite.

Se produciamo programmi da tenere archiviati dobbiamo scriverli con un editor di testo (che non sia un word processor) e salvarli con estensione `.pl6`.
Le istruzioni che compongono i programmi devono terminare con punto e virgola (`;`).
Posizionati nella directory dove abbiamo salvato il file lanciamo il comando
`rakudo` <nome_file>`.pl6`
ed eseguiamo il programma.

Nelle primissime versioni di Perl 6 in luogo del comando `rakudo` si usava il comando `perl6`. Attualmente, non so fino a quando, possiamo usarli entrambi alternativamente.

3 Tipi

Distinguo i tipi di dato utilizzabili con Raku tra tipi scalari, costituiti da un singolo valore, e tipi contenitore.

3.1 Tipi scalari

Sono i soliti che troviamo in tutti i linguaggi di programmazione.

3.1.1 Numeri

Raku tratta numeri interi e numeri in virgola mobile (dotati del separatore decimale `.`).

I numeri interi (`Int`) sono a precisione arbitraria, cioè possono avere dimensione tanto grande quanto tollerata dall'hardware su cui lavoriamo.

I numeri in virgola mobile (`Rat`) sono a precisione limitata a 18 cifre, virgola compresa.

3.1.2 Stringhe

La stringa (`Str`) è una sequenza immutabile di caratteri racchiusa tra apici semplici (') o doppi (").

La stringa racchiusa tra doppi apici ha il privilegio di poter essere interpolata, cioè di avere inserito il contenuto di una variabile in essa nominata.

3.1.3 Valori booleani

I valori booleani (`Bool`) sono `True` (vero) e `False` (falso).

3.2 Tipi contenitori

Sono insiemi variamente organizzati di elementi di tipo scalare o di tipo contenitore.

L'elemento può essere indicato letteralmente o attraverso il richiamo di una variabile che lo contiene.

3.2.1 Lista

La lista (`List`) è una sequenza immutabile di elementi separati da virgola e racchiusi tra parentesi tonde.
`(78, "pippo", 22.6)` crea una lista contenente un intero, una stringa e un numero razionale.

3.2.2 Array

L'array (`Array`), altrimenti detto vettore, è una collezione ordinata di elementi separati da virgola e racchiusi tra parentesi quadre.

A differenza di quanto accade per la Lista, gli elementi dell'Array possono essere aggiunti e tolti.
`[32, 'u', (17, 45.8)]` crea un array contenente un intero, una stringa e una lista.

3.2.3 Hash

L'hash (`Pair`), altrimenti detto dizionario, è una sequenza di elementi contraddistinti da una chiave accoppiata a ciascun elemento.

Le accoppiate si creano con il simbolo `=>` e le coppie si separano con virgole (`,`).
`'Italia' => 'Roma', 'Francia' => 'Parigi'` crea un hash che associa Stati e relative Capitali.

4 Variabili

Le variabili sono delle locazioni di memoria, delle scatole, alle quali diamo un nome, destinate a contenere valori di un certo tipo.

In Raku le variabili si creano nel momento in cui servono, senza bisogno, come avviene in molti linguaggi di programmazione, di dichiararle prima.

Di norma la tipizzazione della variabile avviene in maniera dinamica automaticamente al momento dell'assegnazione del valore, in quanto Raku riconosce il tipo da come scriviamo il valore stesso.
In questo modo una variabile inizializzata con un numero nel corso del programma può, per esempio, essere utilizzata per contenere una stringa.

L'assegnazione del valore viene fatta con l'operatore = e la sintassi
`my <nome_variabile> = <valore>`.
<nome_variabile> deve avere come primo carattere un sigillo, cioè un carattere prefisso che convenzionalmente è
`$` per variabili contenenti un valore scalare o una lista,
`@` per variabili contenenti un array,
`%` per variabili contenenti un hash.
`my $nome = "Vittorio"` crea la variabile `nome` e le assegna il valore di tipo stringa `Vittorio`
`my $raggio = 6` crea la variabile `raggio` e le assegna il valore di tipo intero `6`
`my @A = [15, 22]` crea l'array (vettore riga) `A` assegnandogli i valori `15 e 22`

Con il comando
<nome_variabile>`.WHAT`
possiamo verificare il tipo di una variabile.

La sintassi che utilizziamo per questa verifica è tipica della programmazione per oggetti e mi dà modo di introdurre il concetto che in Raku tutto è un oggetto.

Nel caso specifico `WHAT` è un metodo dell'oggetto <nome_variabile> e viene richiamato con il classico punto (`.`).

Addirittura un numero che digitiamo sulla tastiera diventa subito un oggetto di Raku.
Se nel REPL digitiamo
`7.WHAT`
abbiamo immediatamente la risposta
`(Int)`
in quanto il 7 che abbiamo digitato è stato immediatamente trasformato da Raku in un oggetto dotato del metodo `.WHAT` che ne indica il tipo.

Può accadere di dover modificare il tipo del dato contenuto in una variabile, per esempio per poter applicare ad esso metodi che non sarebbero propri del tipo assegnato dinamicamente alla variabile che lo contiene.

I casi più frequenti sono quelli di dover convertire in stringa un valore numerico e viceversa.

Per fare questo ricorriamo ai metodi di cui è dotato l'oggetto Variabile, metodi semplicemente identificati con il simbolo del tipo che ci interessa (`Int`, `Rat`, `Str`).
Se la variabile `$n` contiene un numero in formato stringa, per leggerne il contenuto in formato intero basta dare il comando `$n.Int`.

* * *

Quando dichiariamo una variabile in Raku possiamo anche evitare la tipizzazione dinamica di default e optare per la tipizzazione statica, quella che consente di stabilire a priori che tipo di dato debba contenere una variabile.

In tal caso creiamo la variabile con la sintassi
`my <tipo> <nome_variabile>`
oppure, nel caso vogliamo anche inizializzarla,
`my <tipo> <nome_variabile> = <valore>.`

La variabile così dichiarata non potrà che contenere, in tutto il corso del programma, che un valore del tipo indicato al momento della sua creazione.
Ad esempio, una variabile creata con
`my Int $x`
non potrà che contenere un numero intero e il tentativo di inserirvi un valore di un tipo diverso darà errore di compilazione.

5 Operatori

Gli operatori collegano tra loro operandi in espressioni che forniscono un risultato.

5.1 Operatori aritmetici

In ordine di precedenza di esecuzione sono i seguenti:
`**` per l'elevamento a potenza,
`*` per la moltiplicazione,
`/` per la divisione,
`div` per la divisione intera (con arrotondamento per difetto)
`%` per il modulo (resto della divisione intera),
`+` per la somma,
`-` per la sottrazione,
`~` per la concatenazione di stringhe[2].

5.2 Operatori di confronto

Servono per confrontare due valori e il risultato che restituiscono è un valore booleano. Sono i seguenti:
`==` uguaglianza tra numeri,
`eq` uguaglianza tra stringhe,
`!=` disuguaglianza tra numeri,
`ne` disuguaglianza tra stringhe,
`<` minore (sempre preceduto da uno spazio),
`lt` minore tra stringhe,
`<=` minore o uguale,
`le` minore o uguale tra stringhe,
`>` maggiore,
`gt` maggiore tra stringhe,

[2]Il carattere ~, detto tilde, si scrive in Linux combinando il tasto AltGr con ì, in Windows combinando il tasto Alt con 126 della tastiera numerica e in Mac combinando il tasto Alt con 5 della tastiera numerica.

`>=` maggiore o uguale,
`ge` maggiore o uguale tra stringhe.
Gli operandi assoggettati al confronto devono avere lo stesso tipo.

5.3 Operatori logici

Forniscono come risultato un valore booleano e sono i seguenti:
`&&` per l'AND logico,
`||` per l'OR logico.

6 Interattività con l'utente

L'interfacciamento con l'utente avviene attraverso il terminale.

6.1 Output

I comandi per l'output sono `say` e `print` con la sintassi
`say` <cosa_scrivere>
`print` <cosa_scrivere>
dove <cosa_scrivere> può essere indicato con una stringa, una espressione matematica o il nome di una variabile che contiene il valore che vogliamo scrivere, anche combinati tra loro separati con una virgola (`,`).

La differenza tra i due è che il primo scrive e va a capo mentre il secondo scrive senza andare a capo.

Esempi:
`say "ciao"` scrive `ciao` e va a capo
`print 5` scrive `5` e non va a capo,
`print 3*7.5` scrive `22.5` e non va a capo,
`say "3 per 8 fa ", 3 * 8` scrive `3 per 8 fa 24` e va a capo,
data la variabile `$x` contenente il valore `16`,
`print $x` scrive `16` e non va a capo,
`say "la variabile x vale $x"` scrive `la variabile x vale 16` e va a capo.
Quest'ultima istruzione, scritta in forma interpolata, potremmo anche scriverla così:
`say "la variabile x vale ", $x`

Se desideriamo formattare l'output abbiamo a disposizione un terzo comando: `printf`, ereditato pari pari dal linguaggio C.

Il comando scrive l'output formattandolo secondo le indicazioni fornite attraverso le così dette direttive di formattazione, che funzionano come dei segnaposto nella stringa che contiene quanto va scritto.

Una direttiva di formattazione si apre con il simbolo % cui fanno seguito, per citare quelli di più ricorrente uso, i seguenti altri simboli:
`s` ad indicare l'inserimento di una stringa,
`d` ad indicare l'inserimento di un numero intero non particolarmente formattato,
`f` ad indicare l'inserimento di un numero decimale non particolarmente formattato,
<n>`d` con n numero intero, ad indicare l'inserimento di un intero allineato a destra in un campo di n caratteri,
`.`<n>`f` con n numero intero, ad indicare l'inserimento di un decimale con una parte decimale di n caratteri (l'ultimo dei quali arrotondato).

Il comando non inserisce il new line e, per andare a capo, occorre terminare la stringa con il carattere di escape `\n`.

Alla fine della stringa, dopo una virgola, si elencano i letterali numerici o le espressioni numeriche (e ne verrà scritto il risultato) o i nomi di variabili (e ne verrà scritto il valore) che dovranno occupare i segnaposto, ponendo il tutto nell'ordine dei segnaposto stessi.

Esempio:
Supponiamo di avere tre variabili: $a intera 78, $b intera 3456 e $c decimale 786,89567.
In questa zona di terminale passiamo due istruzioni con `printf` e ne vediamo l'effetto:

```
> printf "2 moltiplicato 3.5 fa %.2f e la variabile c vale %.3f\n", 2*3.5, $c
2 moltiplicato 3.5 fa 7.00 e la variabile c vale 786.896
> printf "la variabile a vale %6d\nla variabile b vale %6d\n", $a, $b
la variabile a vale     78
la variabile b vale   3456
```

6.2 Input

Il comando per l'input è `get`.

Passato il comando, l'esecuzione del programma resta sospesa fino a quando l'utente non ha inserito da tastiera un dato.

Eventuali istruzioni su che cosa l'utente debba inserire vanno impartite con un precedente messaggio scritto con `say` o `print`.

Il dato inserito viene normalmente letto come stringa. Per leggerlo altrimenti dobbiamo invocare il metodo adeguato:
`get.Int` legge il dato come numero intero,
`get.Rat` lo legge come numero razionale.

Se ci accontentiamo di leggere il dato come stringa possiamo utilizzare una buona combinazione tra `say` e `print` rappresentata dal comando `prompt`, che viene scritto seguito da una stringa contenente la descrizione di quanto si vuole che l'utente inserisca.
`my $x = get` inserisce quanto digitato nella variabile `$x` come stringa,
`my $y = get.Rat` inserisce quanto digitato nella variabile `$y` come numero decimale,
`my $nome = prompt "Come ti chiami? "` chiede il nome all'utente e lo inserisce nella variabile `$nome`.

7 Strutture di controllo

Come ogni altro linguaggio di programmazione, Raku ha dei comandi per assoggettare l'esecuzione di certe istruzioni al verificarsi di determinate condizioni oppure per la ripetizione dell'esecuzione di una o più istruzioni.

7.1 Esecuzione condizionale

if

L'istruzione `if`, chiamata istruzione di esecuzione condizionale (in inglese if è il nostro se), ci dà modo di assoggettare l'esecuzione di un blocco di istruzioni al verificarsi di una determinata condizione: se la condizione è vera viene eseguito il blocco di istruzioni, altrimenti si prosegue l'esecuzione del programma saltando il blocco stesso.

La sintassi è la seguente

```
if (<condizione>)
{
<istruzioni>
}
```

dove <condizione> è una qualsiasi espressione che relaziona due valori attraverso operatori di confronto: se la condizione si verifica vengono eseguite la o le istruzioni indicate tra le parentesi graffe, altrimenti si passa oltre (parentesi tonde non necessarie se l'espressione è semplice).

L'istruzione `if` si presta anche all'esecuzione condizionale a due rami. Per ottenere questo dobbiamo abbinarla all'istruzione `else` con questa sintassi

```
if (<condizione>)
{
<istruzioni>
}
else
{
<istruzioni>
}
```

In questo caso se la condizione si verifica vengono eseguite le istruzioni contenute nel primo blocco, prima della parola chiave `else`, altrimenti vengono eseguite quelle contenute nel secondo blocco dopo `else` (altrimenti). In ogni caso proseguendo poi nell'esecuzione del resto del programma.

Possiamo infine gestire l'esecuzione condizionale a più rami abbinando a `if` l'istruzione `elseif` con questa sintassi

```
if (<condizione>)
{
<istruzioni>
}
elseif <condizione>
{
<istruzioni>
}
elseif <condizione>
{
<istruzioni>
}
else
{
<istruzioni>
}
```

Gli `elseif` possono essere quanti vogliamo.

Le parentesi graffe di apertura dei blocchi possono essere poste alla fine della riga che contiene la condizione, purché distanziate di uno spazio rispetto a ciò che precede.

given

Condiziona l'esecuzione di certe istruzioni al verificarsi di ricorrenze legate a un valore con la sintassi

```
given <valore>
{
when <ricorrenza> {<istruzioni>}
when <ricorrenza> {<istruzioni>}
...
default {<istruzioni>}
}
```

<valore> può essere espresso in qualsiasi modo, anche richiamando una variabile che lo contiene.
<ricorrenza> può essere espresso come valore o come caratteristica di un valore.
Al primo verificarsi di una ricorrenza si interrompe il processo di confronto.
Se si vuole che questo continui dopo un confronto occorre aggiungere alle istruzioni relative a questo confronto, separata da punto e virgola (;) , l'istruzione `proceed`.

Questa istruzione condizionale ricorda l'istruzione `switch` presente in altri linguaggi ma è molto più potente.

Ad esempio, data una variabile `$giorno` contenente la stringa `"Martedì"`, il seguente codice

```
given $giorno
{
when Str { say "è una stringa"; proceed }
when "Lunedì" {say "Primo giorno della settimana"}
when "Martedì" {say "Secondo giorno della settimana"}
when "Mercoledì" {say "Terzo giorno della settimana"}
default { say "non ho capito" }
}
```

ritorna

```
è una stringa
Secondo giorno della settimana
```

7.2 Ripetizione

while

Si usa quando si vuole ripetere l'esecuzione di una istruzione o di un blocco di istruzioni per un numero definito di volte.

La sintassi per ripetere `n` volte un blocco di istruzioni è:

```
my $i = 0
while $i < n
{
   <istruzioni>
   $i = $i + 1
}
```

for

Completamente modificata rispetto a Perl 5, si usa per reiterare istruzioni legate agli elementi contenuti in un array.

La sua applicazione più semplice consiste nel legare la ripetizione di una istruzione al numero di elementi di un array.
In questo caso, data l'esistenza di una variabile `@a` di `n` elementi, con l'istruzione

```
for @a
{
    <istruzioni>
}
```

otteniamo la ripetizione delle istruzioni n volte (`n` essendo il numero degli elementi di `@a`).
In questo caso abbiamo semplicemente un altro modo di ottenere quanto possiamo ottenere utilizzando `while`.

Più interessante l'applicazione per ripetere una istruzione su ciascun elemento di un array.
In questo caso, data l'esistenza di una variabile `@a`, con l'istruzione

```
for @a -> $elemento
{
    <istruzioni_per_elemento>
}
```

otteniamo la ripetizione delle istruzioni su ciascun elemento dell'array.
Per ottenere questo dobbiamo preventivamente creare la variabile intermedia, che ho chiamato `$elemento`, accedendo a ciascun elemento dell'array con l'operatore freccia (`->`)

Esempio:
Dato l'array `@numeri` formato dai numeri `2, 4 e 6`, con l'istruzione

```
for @numeri -> $e
{
   say $e ** 2
}
```

otteniamo la stampa di

```
4
16
36
```

8 Semplici programmi con paradigma imperativo

Quanto visto finora ci consente di creare i nostri primi programmi secondo un paradigma imperativo, il più semplice, consistente in una successione di istruzioni che il computer esegue, una via l'altra.

Cominciamo da questo, banalissimo, che chiede il nome all'utente e gli rivolge un saluto personalizzato:

```
my $nome = prompt "Come ti chiami?  ";
say "Ciao, ", $nome, "!";
```

Quest'altro calcola la circonferenza e l'area di un cerchio chiedendone il raggio all'utente:

```
say "Raggio del cerchio?";
my $r = get.Rat;
say "Circonferenza:  ", 2 * $r * pi;
say "Area:  ", $r * $r * pi;
```

Ricordo che i valori delle due più famose costanti matematiche, π ed e, sono richiamabili, rispettivamente, con i simboli `pi` ed `e`.

Se nel listato del programma vogliamo inserire commenti possiamo farlo su una sola riga anteponendo a commento il simbolo #.

9 Funzioni

La funzione, altrimenti detta subroutine, metodo o procedura, è una sezione di programma in cui è racchiusa una serie di istruzioni da eseguire e che vengono eseguite ogni volta che la funzione è chiamata.

Per semplificare la scrittura di un certo programma, potrebbe essere utile raggruppare alcune istruzioni all'interno del programma stesso, creando funzioni richiamabili, in modo da suddividere i compiti e da evitare di scrivere più volte le stesse cose.

La sintassi per definire una funzione è
`sub` <nome_funzione>
{
<istruzioni>
}
dove <nome_funzione> è il nome che intendiamo dare alla funzione, ed è il nome che useremo per richiamarla.
Se definisco la funzione

```
sub salutami {say 'Ciao, Vittorio!'}
```

che, se è semplice, possiamo fare su una sola riga,
ogni volta che in un programma scriverò l'istruzione `salutami` verrò salutato con la scritta `Ciao, Vittorio!`.

Molto vantaggioso può risultare disporre di funzioni che producano risultati diversi lavorando su parametri indicati quando si richiamano le funzioni stesse. In questo caso la sintassi per definire la funzione diventa

```
sub <nome_funzione> ( <parametro>, <parametro>, ...)
{
<istruzioni>
}
```

dove <parametro> viene indicato con il nome di una variabile, volendo, preceduto da un tipo che ne condizioni il contenuto.
Se definisco la funzione

```
sub saluta (Str $nome) {say 'Ciao, ' ~ $nome ~ '!'}
```

scrivendo `saluta 'Pietro'` saluterò Pietro e scrivendo `saluta 'Giovanni'` saluterò Giovanni.
Se definisco la funzione

```
sub area_triangolo ($base, $altezza) {$base * $altezza / 2}
```

scrivendo

```
area_triangolo 12, 5
```

otterrò il valore 30, che è l'area del triangolo di base 12 e altezza 5.

9.1 Funzioni preconfezionate di base

Raku ci offre tutta una serie di funzioni utili per fare tante cose senza che dobbiamo invocare alcuna libreria per poterle utilizzare.

Alcune le conosciamo già.

Pensiamo alle funzioni `print`, `say` e `printf` per l'output, alla funzione `get` per l'input.

Molto utili sono quelle di più ricorrente utilità in campo matematico.

9.1.1 Funzioni aritmetiche

`abs n` restituisce il valore assoluto del valore numerico n,
`exp n` restituisce la potenza ennesima del numero e,
`round n` restituisce, sotto forma di numero intero, la parte intera di n arrotondata,
`log n` restituisce il logaritmo naturale di n,
`log10 n` restituisce il logaritmo decimale di n,
`sqrt n` restituisce la radice quadrata di n,
`pi` restituisce il valore della costante π,
`e` restituisce il valore della costante e.

9.1.2 Funzioni trigonometriche

Esprimendo n in radianti abbiamo le funzioni dirette `sin n`, `cos n` e `tan n`.
Le funzioni inverse `asin n`, `acos n` e `atan n` restituiscono l'angolo in radianti.

10 Semplici programmi con paradigma funzionale

Secondo il paradigma funzionale, i due programmini del Capitolo 8 possono essere scritti in questo modo.

Il primo, che chiede il nome all'utente per salutarlo:

```
sub saluta (Str $nome)
{
    say "Ciao, $nome!";
}
saluta "Giorgio";
```

Il secondo, che calcola circonferenza e area del cerchio noto il raggio:

```
sub circonferenza ($raggio)
{
    $raggio*2*pi;
}
sub area ($raggio)
{
    $raggio**2 * pi;
}
say "Circonferenza:  ", circonferenza 6;
say "Area:  ", area 6;
```

In questi casi l'esecuzione del codice non avviene una riga dopo l'altra come avveniva nel paradigma imperativo. Il codice che definisce funzioni, che viene prima delle istruzioni destinate ad eseguirlo, viene eseguito solo quando e se le funzioni sono richiamate.

I parametri delle funzioni sono inseriti dal programmatore ma potrebbero essere richiesti all'utente secondo quanto abbiamo già imparato.

Le indentazioni non sono richieste ma possono servire per leggere meglio il listato del programma.

11 Oggetti

Ho già avuto modo di ricordare che in Raku tutto è un oggetto.

Un oggetto informatico è un insieme di attributi e di metodi: gli attributi definiscono le caratteristiche dell'oggetto e i metodi definiscono le operazioni eseguibili sull'oggetto.

Molte delle funzioni che abbiamo visto fin qui, oltre che essere funzioni del linguaggio Raku, sono anche metodi degli oggetti Raku.

Per scrivere `Ciao, Vittorio!` abbiamo visto che esiste nel linguaggio la funzione `say` alla quale passiamo il parametro stringa `'Ciao, Vittorio!'`.
Ma allo stesso risultato perveniamo scrivendo
`'Ciao, Vittorio!'.say`
cioè scrivendo la stringa `'Ciao, Vittorio!'` e richiamandone il metodo `say` per scriverla.

Tutte le funzioni aritmetiche elencate nel paragrafo 9.1.1 e le funzioni trigonometriche elencate nel paragrafo 9.1.2 sono anche disponibili come funzioni membro dell'oggetto n. Sicché, per esempio, il logaritmo naturale di n possiamo ottenerlo con `n.log`, il valore assoluto di -2 possiamo ottenerlo con `(-2).abs`, il seno di $\pi/2$ possiamo ottenerlo con (pi/2).sin, ecc.
In questo contesto, se n è rappresentato da una espressione, l'espressione va racchiusa tra parentesi tonde (e il numero negativo -2 è considerato una espressione, come pi/2).

Altre funzioni aritmetiche che troviamo solo come membri dell'oggetto n sono
`n.Int` restituisce la parte intera di un numero razionale eliminando la parte dopo la virgola,
`n.rand` genera un numero casuale compreso tra 0 e n,
`n.rand.Int` genera un numero intero casuale compreso tra 0 e n.

Abbiamo anche visto che esiste la funzione membro `.WHAT` per individuare il tipo di un oggetto qualsiasi.

Funzioni che ritroviamo unicamente come funzioni membro di oggetti sono quelle che ci consentono di lavorare con stringhe e array.

funzioni membro di oggetti stringa

`.chars` ritorna il numero dei caratteri di una stringa,
`.index(<carattere>)` ritorna il posto occupato da un carattere nella stringa,
`.uc` ritorna la stringa in caratteri tutti maiuscoli,
`.lc` ritorna la stringa in caratteri tutti minuscoli,
`.substr(n)` estrae una sotto-stringa dal carattere di indice n alla fine,
`.substr(a..b)` estrae una sotto-stringa dal carattere di indice a al carattere di indice b.

Ricordo che gli indici dei caratteri partono da 0.

Ricordo altresì che la stringa è immutabile, per cui estrarre un carattere da una stringa non significa cancellarlo ma estrarlo come valore, per stamparlo, per inserirlo in una variabile, ecc.

funzioni membro di oggetti array

.elems ritorna il numero degli elementi di un array (funzione disponibile anche per la lista)
.push(<elemento>, <elemento>, ...) aggiunge elementi in un array,
.pop rimuove l'ultimo elemento di un array e lo restituisce,
.splice(a,n) rimuove n elementi iniziando da a e li restituisce.

Sempre ricordando che l'indicizzazione della posizione degli elementi parte da 0.

Gli elementi restituiti possono essere utilizzati per essere stampati, inseriti in al tre variabili, ecc.

11.1 Come si crea un oggetto

Per costruire un oggetto dobbiamo innanzi tutto disporre di un prototipo, la classe, che prevede come è fatto l'oggetto che intendiamo costruire

```
class <identificatore>
{
    has $.<identificatore>;
    has $.<identificatore>;
    ....
    method <identificatore>
    {
        <istruzioni>
    }
    method <identificatore>
    {
        <istruzioni>
    }
    ....
}
```

La parola chiave `class` definisce la classe, la parola chiave `has` definisce gli attributi e la parola chiave `method` definisce i metodi, detti anche funzioni membro.

La costruzione dell'oggetto avviene con la sintassi
`my $`<identificatore_oggetto> `=` <identificatore_classe>`.new(`<valore_attributi>`)`
dove <valore_attributi> si indica con la sintassi
<identificatore_attributo> `=>` <valore>

12 Semplice programma con paradigma a oggetti

Dei due programmini che abbiamo visto nei Capitoli 8 e 10, quello destinato a formulare un saluto è troppo banale per disturbare la programmazione a oggetti, specie se utilizziamo un linguaggio, come Raku, che ci dà modo di sviluppare con paradigma imperativo o funzionale.

Potremmo farlo, ma il gioco non vale la candela.

Del resto chi conosce il linguaggio Java, dove si programma solo a oggetti, sa quanto sia relativamente complicato scrivere, con quel paradigma, un programmino che chiede il nome all'utente per salutarlo.

Come esempio di programmazione a oggetti propongo invece la riscrittura del programma per il calcolo della circonferenza e dell'area di un cerchio che abbiamo già visto secondo il paradigma imperativo e secondo il paradigma funzionale.

```
class cerchio
{
    has $.raggio;
    method circonferenza
    {
        $.raggio * 2 * pi;
    }
    method area
    {
        $.raggio ** 2 * pi;
    }
}
say "raggio del cerchio?  ";
my $r = get.Rat;
my $cerchio = cerchio.new(raggio => $r);
say "Per un cerchio di raggio $r";
say "La circonferenza è ", $cerchio.circonferenza;
say "L'area è ", $cerchio.area;
```

Abbiamo scritto la classe prototipo per creare un cerchio avente un certo raggio e due metodi per calcolare circonferenza e area.
Poi abbiamo chiesto all'utente di indicare un raggio.
Abbiamo poi costruito un oggetto cerchio avente il raggio indicato dall'utente.
Infine abbiamo determinato circonferenza e cerchio richiamando i relativi metodi dell'oggetto cerchio che abbiamo costruito con quel determinato raggio.

13 Lavorare con file

Per lavorare con i file Raku ci presenta un metodo assolutamente innovativo rispetto ad altri linguaggi e allo stesso linguaggio Perl precedente a Perl6, poi denominato Raku.

Tutto si basa su due funzioni, `slurp` e `spurt`.

slurp

Legge dati da un file di testo con la sintassi
`slurp "<nome_file>";`
dove <nome_file> rappresenta il percorso ad un file esistente.

spurt

Scrive dati su un file, creandolo se non esiste, con la sintassi
`spurt "<nome_file>", <stringa_dato>;`
dove
<nome_file> rappresenta il percorso ad un file esistente o da creare: se il file esiste viene sovrascritto.
<stringa_dato> è il dato da inserire nel file, scritto come stringa tra virgolette. Per inserire dati incolonnati occorre terminare la stringa con `\n`.

Per evitare la sovrascrittura del file ed inserire il dato dopo quelli già esistenti nel file la sintassi è
`spurt "<nome_file>", <stringa_dato> , :append;`

14 Moduli

I Moduli sono librerie disponibili nel repository di Raku, raggiungibile all'indirizzo *https://modules.perl6.org/*.

Vi troviamo un indice alfabetico dei moduli esistenti e, cliccando su quello che ci interessa, possiamo vedere di cosa si tratta esattamente e vedere le istruzioni per il suo utilizzo.

Per poter essere utilizzato, il modulo deve essere installato e, per questo è disponibile la funzione `zef` con la sintassi
`zef install` "nome_del_modulo".

Una volta installato, per richiamare le funzioni che esso contiene occorre dichiararne l'uso all'inizio del programma con la sintassi
`use` "nome_del_modulo".

Interessanti per dilettanti e studenti possono essere i moduli
`DB::SQLite` per lavorare con database SQLite,
`Math::Matrix` per lavorare con matrici e algebra lineare,
`Math::Libgsl::Statistics` per disporre di innumerevoli funzioni statistiche.

15 Lavorare con il testo

Ciò che abbiamo visto sin qui mostra Raku come un linguaggio di programmazione all purpose che ci può aiutare in molti casi, non da ultimo nel calcolo su grandi numeri, grazie al trattamento degli interi in precisione arbitraria che ci evita il fastidioso problema degli overflow.

Ma per l'erede del linguaggio Perl (Practical Extraction and Report Language) non sarebbe giusto non soffermarci anche sulla sua particolare predisposizione al trattamento del testo, che fu il primo obiettivo del linguaggio.

Premessa indispensabile per proseguire è almeno una infarinatura sulla tecnica delle espressioni regolari (Regex).

15.1 Cenni sulle espressioni regolari

Una espressione regolare è una sequenza di caratteri, racchiusi tra due barre oblique (slash), che, secondo una precisa sintassi, ci permette di descrivere uno schema, un pattern, di stringa.

L'argomento non è facile e qui mi limito alle basi del discorso. Per approfondimenti abbiamo la documentazione ufficiale all'indirizzo *https://docs.raku.org/language/regexes*.

Il tipo più semplice di espressione regolare per definire una stringa è quello che ne elenca i caratteri:
`/pippo/` è una espressione regolare che definisce la stringa "pippo", cioè una stringa composta dai caratteri `p`, `i`, `p`, `p` ed `o`.

Più che di uno schema di stringa, si tratta, in questo caso, della stringa stessa in quanto abbiamo utilizzato gli stessi caratteri che la compongono.

Ma le espressioni regolari si avvalgono anche di caratteri speciali utilizzando i quali non definiamo la stringa ma la descriviamo.

La stringa "pippo" ha, per esempio, la caratteristica di essere composta da lettere dell'alfabeto. L'espressione regolare che la descrive in quanto tale è `/\w+/`: con il carattere speciale `\w` seguito dal `+` intendiamo dire che la stringa è composta da uno o più caratteri alfanumerici.

Per descrivere la stringa "1234" usiamo l'espressione regolare `/\d+/`, dove il carattere speciale `\d` indica che si tratta di una stringa composta solo da cifre.

Per descrivere la stringa contenente un indirizzo email valido usiamo l'espressione regolare
`/\w+\@\w+\.\w+/`
nella quale diciamo che lo schema di una stringa contenente un indirizzo email valido consiste in uno o più caratteri alfanumerici seguiti dal carattere @, a sua volta seguito da uno o più caratteri alfanumerici separati da un punto.

Nella seguente tabella indico i principali termini per la composizione di una espressione regolare:

simbolo	significato
`pippo`	la stringa alfanumerica "pippo"
`i`	il carattere alfanumerico (stringa) "i"
`4`	il carattere alfanumerico (stringa) "4"
`.`	un qualsiasi carattere
`\d`	una cifra qualsiasi
`\w`	un carattere alfanumerico
`\W`	un carattere non alfanumerico
`\s`	un carattere di spaziatura
`\n`	il carattere new line
`\@`	il carattere @
`?`	contrassegna un carattere ripetuto 0 o una volta
`*`	contrassegna un carattere ripetuto 0 o più volte
`+`	contrassegna un carattere ripetuto una o più volte
`**n`	contrassegna un carattere ripetuto n volte esatte
`**n..m`	contrassegna un carattere ripetuto da n a m volte
`^`	la stringa inizia con l'espressione seguente
`$`	la stringa termina con l'espressione precedente

e propongo alcuni esempi per capire:
. `/^3\d**9/` indica lo schema di un numero telefonico mobile italiano (inizia con il numero 3 e prosegue con una sequenza di nove cifre);
. `/\w+2$/` indica lo schema di una stringa di caratteri alfanumerici terminante con il numero 2;
. `/\w+\s2$/` indica lo schema di una stringa di caratteri alfanumerici terminante con il numero 2 preceduto da uno spazio;
. `/^\d\w**3..5\d$/` indica lo schema di una stringa che comincia con una cifra, prosegue con da 3 a 5 caratteri alfanumerici e termina con una cifra.

15.2 Elaborazioni sulle stringhe

Parlando dei tipi di dato abbiamo visto il tipo Stringa (`Str`), definendolo come sequenza immutabile di caratteri racchiusa tra apici semplici (') o doppi ("): quella tra caratteri doppi come stringa interpolabile, cioè costruibile richiamando al suo interno il contenuto di una o più variabili.

La delimitazione della stringa, oltre che utilizzando gli apici, può avvenire utilizzando gli operatori di delimitazione `q` (per l'equivalente della stringa tra apici semplici) e `qq` (per l'equivalente della stringa tra doppi apici) e racchiudendo i caratteri della stringa tra i delimitatori `|` e `|`, oppure `[` e `]`, oppure `{` e `}`.
Esempi:
`my $miastringa = q|Vittorio|` crea la stringa `Vittorio` e la inserisce nella variabile `$miastringa`
`qq {Ciao $miastringa}` crea la stringa interpolata `Ciao Vittorio`.
In questo modo possiamo costruire stringhe contenenti qualsiasi tipo di apici, senza che la presenza di questi funga da delimitatore.

Altro utile operatore di manipolazione del testo è l'operatore `qw` con il quale possiamo creare una lista di parole, racchiudendole tra i delimitatori `|` e `|`, oppure `[` e `]`, oppure `{` e `}`, e inserirla in una variabile di tipo lista o in una variabile di tipo array.
Esempi:
`qw |Pippo Pluto Paperino|` crea la lista `(Pippo Pluto Paperino)`
`my $l = qw |Pippo Pluto Paperino|` inserisce le parole in una variabile di tipo lista,
`my @a = qw |Pippo Pluto Paperino|` inserisce le parole in una variabile di tipo array.

Ferma restando la non modificabilità della stringa, possiamo derivare da una stringa una nuova stringa ripulita dell'ultimo carattere della stringa precedente con l'operatore `chop`. La cosa può risultare utile per avere a disposizione la stessa stringa ripulita del carattere di fine riga.
Esempio:
Data la variabile `$s` contenente la stringa `"Pippo\n"`, con
`my $nuovas = chop $s` creiamo una nuova variabile contenente la stringa `"Pippo"`.

Nel Capitolo 11 abbiamo visto le funzioni membro dell'oggetto stringa, funzioni attraverso le quali, salvaguardando l'immutabilità della stringa, possiamo compiere una serie di elaborazioni per derivare da una stringa altre stringhe con caratteristiche diverse.

Ma la potenza di Raku va ben oltre e ci consente di intervenire su una variabile contenente testo per analizzarlo più a fondo, persino per modificarlo.

15.3 Analisi del testo

Per analizzare un testo dobbiamo innanzi tutto inserirlo in una variabile, o scrivendolo in formato stringa
`my $testo = "Ma che bella giornata!";`
o prelevandolo da un file di testo
`my $testo = slurp " <nome_file> ";`
dove <nome_file> rappresenta il percorso ad un file esistente.

In questi casi la variabile `$testo` è di tipo stringa e una sola stringa contiene tutto il testo.

L'analisi viene compiuta sul contenuto della variabile avvalendosi dell'operatore di corrispondenza, detto anche operatore di matching, `~~` (una doppia tilde)[3], posto tra la variabile contenente la stringa di testo ed una espressione regolare: il risultato del matching è un valore booleano, `true` se c'è corrispondenza, `false` se non c'è corrispondenza.

Se la stringa-testo da analizzare è di una sola parola il matching è utile per verificare che la stringa corrisponda ad un certo schema
`$testo ~~ /\d+/` verifica che la stringa contenga numeri,
`$testo ~~ /\w+\@\w+\.\w+/` verifica che la stringa contenga un indirizzo di posta elettronica.

Se la stringa-testo da analizzare è composta da più parole il matching può essere utile per verificare la presenza di una certa sottostringa.
`$testo ~~ /ciao/` verifica che la stringa contenga la sottostringa `ciao`.

Con il seguente programma si chiede l'inserimento di un numero di telefono cellulare italiano e il dato non viene acquisito fino a quando il numero inserito non corrisponde allo schema, al pattern, giusto (prima cifra 3 e poi altre 9 cifre):

```
my $telefono = "";
input;
sub input
{
    $telefono = prompt "Inserisci un numero di cellulare italiano:  ";
    if $telefono ~~ /^3\d**9/
    {
        say "Ho registrato il numero $telefono.";
    }
    else
    {
        say "Il numero inserito non è valido.";
        say "Riprova.";
        input;
    }
}
```

[3]Rammento che il carattere ~, detto tilde, si scrive in Linux combinando il tasto AltGr con ì, in Windows combinando il tasto Alt con 126 della tastiera numerica e in Mac combinando il tasto Alt con 5 della tastiera numerica.

Supponiamo ora di avere una variabile stringa `$testo` contenente la seguente frase
`"Mi sono accasato a Milano ma ho mantenuto anche la casa di Novara"`
e di voler verificare che essa contenga la parola «casa».
Il matching `$testo ~~ /casa/` dà esito positivo, ma non perché si è verificata la presenza della parola «casa» in riferimento a quella di Novara ma semplicemente perché si è innanzi tutto verificata la presenza della ricorrenza «casa» come sottostringa nella parola «accasato».
Per individuare la presenza della parola «casa» dovremmo impostare il matching con l'espressione regolare `/\Wcasa\W/` che indica come sottostringa da verificare il termine casa preceduto e seguito da un carattere non alfanumerico, ad indicare che si deve trattare di una parola isolata (scritta tra due spazi, tra due parentesi, tra uno spazio e un carattere di punteggiatura, ecc.).

Quando il testo di più parole è concentrato in una sola stringa non possiamo fare gran che d'altro.

Se vogliamo sapere quanti numeri di telefono validi o quanti indirizzi di posta elettronica sono contenuti in un testo dobbiamo ricorrere alla funzione `grep`, attraverso la quale possiamo estrarre tutte le ricorrenze vere in modo da poterle elencare e contare.
La funzione `grep` non opera però su variabili di tipo stringa ma su variabili di tipo array. Essa itera su ciascun elemento dell'array stesso e restituisce gli elementi per i quali il matching è `true`, elencandoli in un altro array.
Dobbiamo però innanzi tutto inserire il testo da analizzare in una variabile di tipo array, in modo che ciascuna parola del testo diventi lei una stringa e le stringhe delle parole componenti il testo siano riunite in un array.
Per creare l'array contenente un testo possiamo usare il comando
`my @testo = qw | <parole_del_testo> |;`
Se il testo è contenuto in un file di tipo stringa, creato come tale o alimentato leggendo un file di testo, dobbiamo creare una variabile di tipo array inserendovi ciascuna parola del testo attraverso la splittatura della stringa del testo in corrispondenza degli spazi che dividono le parole con la seguente sintassi
`my @testo = split(/\s/, <variabile_stringa>)`
in modo che ciascuna parola del testo diventi un elemento dell'array.
Dall'array `@testo` estraiamo poi le ricorrenze con il comando
`grep` <espressione_regolare>`, @testo`.

Quanto appena argomentato possiamo verificarlo con questo esempio

```
my @testo = qw|Quando torno a casa mi sento felice:  la mia casa è molto bella.|;
my @corrispondenze = grep /casa/, @testo;
my $n = @corrispondenze.elems;
say "La parola casa è presente $n volte.";
```

Lo script produce il risultato

```
La parola casa è presente 2 volte.
```

Esempio un tantino più complesso.
Questo è il contenuto del file di testo `testo.txt` archiviato nella mia cartella Documenti:

```
«Il miglior modo di comunicare con me è l'indirizzo di posta elettronica
vittorio@vittal.it.  Per comunicare con Giulia è meglio l'indirizzo
giulia.rossi@gmail.com.».
```

Questo programmino verifica quanti indirizzi email contiene il testo e li scrive:

```
my $testo = slurp "/home/vittorio/Documenti/testo.txt";
my @testo = split(/\s/, $testo);
my @ricorrenze = grep /\w+\@\w+\.\w+/, @testo;
my $n = @ricorrenze.elems;
say "Il testo contiene i seguenti $n indirizzi email:";
for @ricorrenze -> $elemento
{
    say $elemento;
}
```

Lo script produce il risultato:

```
Il testo contiene i seguenti 2 indirizzi email:
vittorio@vittal.it.
giulia.rossi@gmail.com.
```

Negli esempi ho utilizzato, per questioni di economia di spazio, testi brevi ma si sappia che Raku non ha praticamente limiti e, con la tecnologia che abbiamo appena visto, potremmo tranquillamente analizzare tutto il testo de I promessi sposi.

Ultimo avvertimento forse di una certa utilità: tutte le ricerche di corrispondenza sono case sensitive, cioè «casa» è diverso da «Casa» e in Raku non c'è più un modo facile di rendere case insensitive le ricerche, come avveniva in edizioni di Perl precedenti a Perl6.
Possiamo ovviare indicando l'alternativa con lo speciale operatore `|` :

```
$testo ~~ /Casa/|/casa/
grep /Casa/|/casa/, @testo
```

riscontrano la presenza indipendentemente dalla minuscola o maiuscola iniziale.

15.4 Manipolazione del testo

Con buona pace per la non modificabilità delle stringhe, Raku ci dà la possibilità di modificarle attraverso la sostituzione di sottostringhe corrispondenti ad uno certo schema con sottostringhe diverse.

L'operazione avviene attraverso l'operatore di matching, `~~` , che abbiamo conosciuto nel precedente paragrafo, collocato tra la stringa da modificare e il modello di sostituzione, modello di sostituzione che viene scritto con la seguente sintassi:
`s |`<espressione_da_sostituire>`|`<espressione_sostituto>`|`
se ci si vuole limitare a sostituire solo la prima occorrenza dell'espressione da sostituire, oppure
`s:g |`<espressione_da_sostituire>`|`<espressione_sostituto>`|`
se si vogliono sostituire tutte le occorrenze dell'espressione da sostituire.

Se abbiamo la stringa `$nome` contenente il valore `"Luigi"` e vogliamo modificarla in `"Luigino"` lo possiamo fare con il comando

```
$nome ~~ s |gi|gino|
```

Se abbiamo la stringa testo `$testo` contenente la frase

```
Quando torno a casa mi sento felice:  la mia casa è molto bella.
```

e vogliamo mettere «casetta» al posto di «casa» usiamo il comando

```
$testo ~~ s:g |casa|casetta|
```

sperando che non vi siano altre sottostringhe «casa» nascoste in parole come «accasamento», in quanto, in questo modo, anche la parola «accasamento» verrebbe convertita in «accasettamento».

Per evitare questo possiamo ricercare la parola da sostituire con una espressione letterale, per esempio così:

```
$testo ~~ s:g |\scasa\s| casetta |
```

ottenendo di sostituire la parola casa se racchiusa tra spazi.
Dal momento che la sottostringa da sostituire contiene anche gli spazi, dobbiamo rimettere questi spazi nell'espressione_sostituto.
Così facendo non andremmo però a sostituire la parola se racchiusa tra parentesi anziché tra spazi o se messa in fine di frase seguita non da uno spazio ma da un carattere di punteggiatura.
Possiamo rimediare con ulteriori passaggi del tipo

```
$testo ~~ s:g |\scasa\,| casetta,|,
```

oppure `$testo ~~ s:g |\scasa\.| casetta.|,`
oppure `$testo ~~ s:g |\(casa\)|(casetta)|`, ecc.

Per dei dilettanti mi fermerei qui. Se c'è un rimedio più elegante per questi problemi non è certo a portata di dilettante.