Programming in Mathematica

Assignments
Definition
Increment, PreIncrement, AddTo, SubtractFrom
Decrement, PreDecrement, TimesBy, DivideBy

Input, InputString
Print
Logical Structures
Clear
Do
EvenQ, OddQ
Nested Loops
While, For
Relational Expressions
If, TrueQ
Logical operators
Transfer of control, Goto

Assignments

Set (=)

lhs = *rhs*

evaluates *rhs* and assigns the result to be the value of *lhs*. From then on, *lhs* is replaced by *rhs* whenever it appears.

$\{l_1, l_2, \ldots\} = \{r_1, r_2, \ldots\}$

evaluates the r_i, and assigns the results to be the values of the corresponding l_i.

- *lhs* can be any expression, including a pattern.
- f[x_] = x^2 is a typical assignment for a pattern. Notice the presence of _ on the left-hand side, but not the right-hand side.
- An assignment of the form *f*[*args*] = *rhs* sets up a transformation rule associated with the symbol *f*.
- New assignments with identical *lhs* overwrite old ones. »
- You can see all the assignments associated with a symbol *f* using ?*f* or Definition[*f*].

Set a value for *x*:

```
In[1]:= x = a + b
Out[1]= a + b
```

```
In[2]:= 1 + x^2
Out[2]= 1 + (a + b)^2
```

Set multiple values:

```
In[1]:= {x, y, z} = Range[3]
Out[1]= {1, 2, 3}

In[2]:= x + y^2 + z^3
Out[2]= 32
```

Ordinary program variables:

```
In[1]:= i = 1;
        While[Prime[i] < 100, i = i + 1]; i
Out[1]= 26
```

Set values for "indexed variables":

```
In[1]:= a[1] = x; a[2] = y;

In[2]:= {a[1], a[2], a[3]}
Out[2]= {x, y, a[3]}
```

Define a function from an expression:

```
In[1]:= Expand[(1 + x)^3]
```

Out[1]= $1 + 3x + 3x^2 + x^3$

```
In[2]:= f[x_] = %
```

Out[2]= $1 + 3x + 3x^2 + x^3$

```
In[3]:= f[a + b]
```

Out[3]= $1 + 3(a + b) + 3(a + b)^2 + (a + b)^3$

Set part of a list:

```
In[1]:= v = {a, b, c, d}
Out[1]= {a, b, c, d}

In[2]:= v[[2]] = x
Out[2]= x

In[3]:= v
Out[3]= {a, x, c, d}
```

```
In[1]:= mat = (1 2 3
               4 5 6
               7 8 9);
```

Replace a row of a matrix:

```
In[2]:= mat[[2]] = mat[[2]] + 10;
        MatrixForm[mat]
```

$$\text{Out[2]//MatrixForm=}\begin{pmatrix}1 & 2 & 3\\ 14 & 15 & 16\\ 7 & 8 & 9\end{pmatrix}$$

Replace a column of a matrix:

```
In[3]:= mat[[All, 3]] = {100, 101, 102};
        MatrixForm[mat]
```

$$\text{Out[3]//MatrixForm=}\begin{pmatrix}1 & 2 & 100\\ 14 & 15 & 101\\ 7 & 8 & 102\end{pmatrix}$$

In[1]:= $x = (a + b)^2$;

In[2]:= 1 + x + 1 / x

$$\text{Out[2]=}\ 1 + \frac{1}{(a+b)^2} + (a+b)^2$$

Set part of an expression:

```
In[1]:= v = 1 + x^5
```

Out[1]= $1 + x^5$

```
In[2]:= v[[2, 2]] = 77777
Out[2]= 77777
```

```
In[3]:= v
```

Out[3]= $1 + x^{77777}$

```
In[1]:= {a, b} = {27, 6};
        While[b ≠ 0, {a, b} = {b, Mod[a, b]}];
        a
Out[1]= 3
```

Find a fixed point:

```
In[1]:= x = 1.0;
        While[Cos[x] ≠ x, x = Cos[x]];
        x
Out[1]= 0.739085
```

Definition

Definition[*symbol*]
prints as the definitions given for a symbol.

- Definition has attribute HoldAll.
- Definition[*symbol*] prints as all values and attributes defined for *symbol*.
- ? *s* uses Definition.
- Definition does not show rules associated with symbols that have attribute ReadProtected.

```
In[1]:= f[x_] := x^2

In[2]:= Definition[f]

Out[2]= f[x_] := x²
```

This gives the definition of symbol itself:

```
In[2]:= symbol = Plus;

In[3]:= Definition[symbol]

Out[3]= symbol = Plus
```

Increment (++)

x++

increases the value of *x* by 1, returning the old value of *x*.

Increment the value by one, and return the old value:

```
In[1]:= k = 1; k++
Out[1]= 1

In[2]:= k
Out[2]= 2
```

```
In[20]:= i = 1;
         While[Prime[i] < 10^6, i++];
         i
Out[22]= 78499

In[23]:= Prime[78499]
Out[23]= 1000003
```

PreIncrement (++)

$++x$

increases the value of x by 1, returning the new value of x.

- $++x$ is equivalent to $x = x + 1$.

Increment the value by 1 and return the new value:

```
In[1]:= k = 1; ++k
Out[1]= 2

In[2]:= k
Out[2]= 2
```

AddTo (+=)

x += dx
adds dx to x and returns the new value of x.

- x += dx is equivalent to $x = x + dx$.

```
In[1]:= k = 1; k += 5
Out[1]= 6

In[2]:= k
Out[2]= 6
```

Add to a numerical value:

```
In[1]:= x = 1.5; x += 3.75; x
Out[1]= 5.25
```

Add to a symbolic value:

```
In[1]:= v = a; v += b; v
Out[1]= a + b
```

Add to all values in a list:

```
In[1]:= x = {1, 2, 3}
Out[1]= {1, 2, 3}

In[2]:= x += 17; x
Out[2]= {18, 19, 20}

In[3]:= x += {20, 21, 22}; x
Out[3]= {38, 40, 42}
```

SubtractFrom (-=)

x -= dx
subtracts dx from x and returns the new value of x.

- x -= dx is equivalent to $x = x - dx$.

```
In[1]:= k = 1; k -= 5
Out[1]= -4

In[2]:= k
Out[2]= -4
```

Subtract from a numerical value:

```
In[1]:= x = 1.5; x -= 0.75; x
Out[1]= 0.75
```

Subtract from a symbolic value:

```
In[1]:= v = a; v -= b; v
Out[1]= a - b
```

Decrement (--)

x--
decreases the value of x by 1, returning the old value of x.

Decrement the value of k by one, but return the old value:

```
In[1]:= k = 1; k--
Out[1]= 1

In[2]:= k
Out[2]= 0
```

Increment and PreIncrement are closely related operations:

```
In[1]:= {a, b, c, d} = {1, 1, 1, 1};
In[2]:= {a++, ++b, c--, --d}
Out[2]= {1, 2, 1, 0}

In[3]:= {a, b, c, d}
Out[3]= {2, 2, 0, 0}
```

PreDecrement (--)

--x
decreases the value of x by 1, returning the new value of x.

- --x is equivalent to $x = x - 1$.

Decrement the value by one and return the new value:

```
In[1]:= k = 1; --k
Out[1]= 0

In[2]:= k
Out[2]= 0
```

The value of i++ is the value of i *before* the increment is done.

```
In[5]:= i = 5; Print[i++]; Print[i]
5
6
```

The value of ++i is the value of i *after* the increment.

```
In[6]:= i = 5; Print[++i]; Print[i]
6
6
```

TimesBy (*=)

x *= c
multiplies x by c and returns the new value of x.

- x *= c is equivalent to $x = x * c$.

```
In[1]:= k = 3;
        k *= 5
Out[1]= 15

In[2]:= k
Out[2]= 15
```

DivideBy (/=)

x /= c
divides x by c and returns the new value of x.

- x /= c is equivalent to $x = x / c$.

```
In[1]:= k = 15;
        k /= 3
Out[1]= 5

In[2]:= k
Out[2]= 5
```

Input

Input[]
  interactively reads in one Wolfram Language expression.

Input[*prompt*]
  requests input, displaying *prompt* as a "prompt".

Input[*prompt*, *init*]
  in a notebook front end uses *init* as the initial contents of the input field.

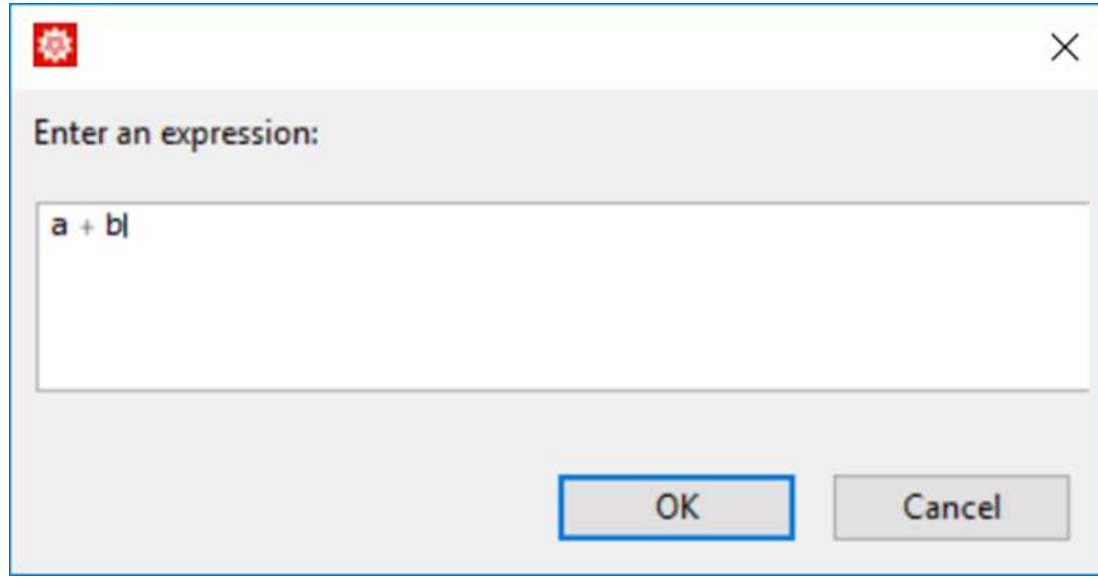
When run this command, Mathematica would **prompt** with a **window.**
Type the symbol and its value.
e.g.

Input["mass = ?"]

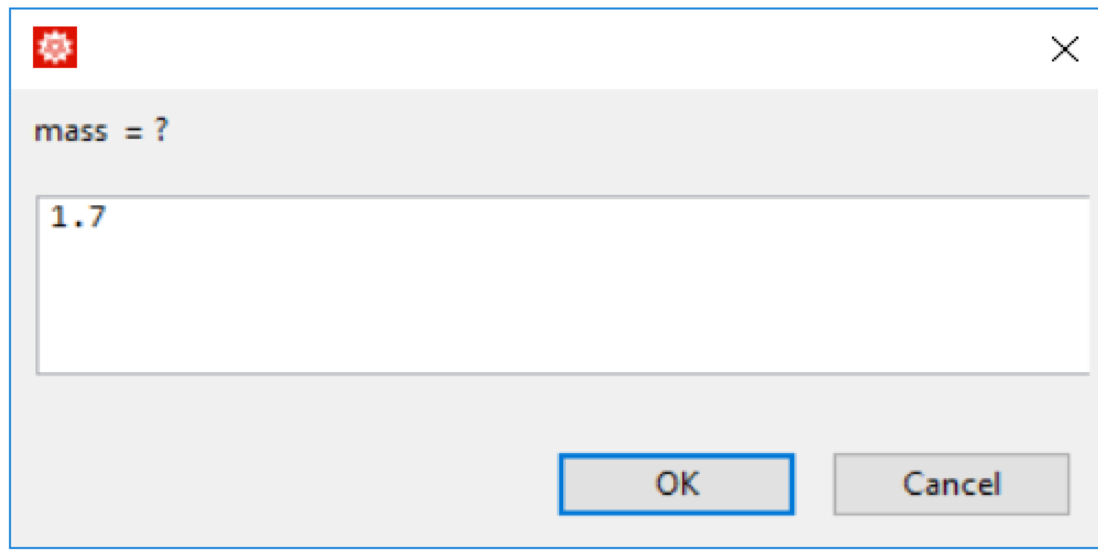
When window appears, type mass = 9.0, and press the enter key.
Mathematica then assigns value 5 to the variable mass, and
responds with the following output:

Out[7]= 9.0

```
Input[]^2
Out[21]= (a + b)^2

In[23]:= x = Input[]
Out[23]= 10
```




```
In[32]:=
    Input["mass  = ?"]

Out[32]= 1.7
```



InputString

InputString[]
  interactively reads in a character string.

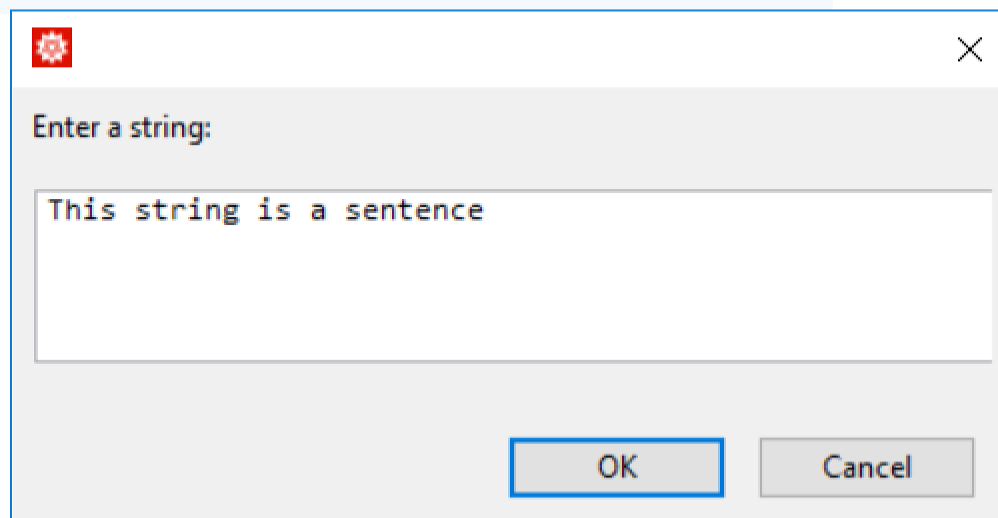
InputString[*prompt*]
  requests input, displaying *prompt* as a "prompt".

InputString[*prompt*, *init*]
  in a notebook front end uses *init* as the initial contents of the input field.

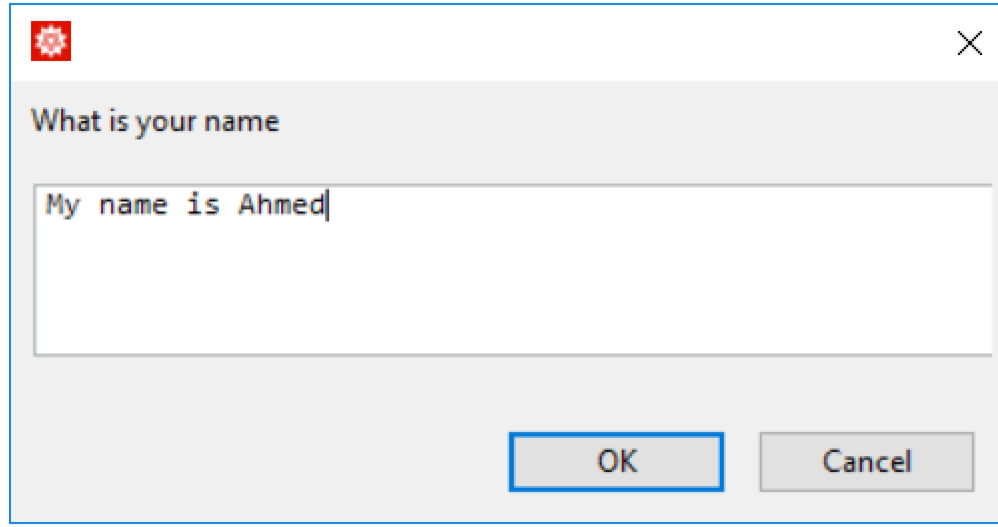
In[33]:=

```
InputString[]
```

Out[33]= `This string is a sentence`



```
In[41]:=
        s = InputString["What is your name"]
Out[41]= My name is Ahmed

In[42]:= s
Out[42]= My name is Ahmed
```



Write a program to find all prime numbers less than a given n.

We will use the loop `While` to find one by one all the prime numbers smaller than n starting from the smallest prime number 2. Notice that here the body of `While` has two sentences.

```
i = 1; n = Input["enter a number"]; pset = {};
While[Prime[i] ≤ n,
  pset = pset ∪ {Prime[i]};
  i++];
pset
```

Output statements

Results obtained in a program are generally written using the following statement:

Print[variable]

Messages can be printed on screen by enclosing them in double quote (**“**) sign:

Print[“You are welcome!”]
You are welcome!

Print x + y, then print a + b:

```
In[1]:= Print[x + y]; Print[a + b]
        x + y
        a + b
```

Print the first 5 primes:

```
In[1]:= Do[Print[Prime[n]], {n, 5}]
        2
        3
        5
        7
        11
```

```
In[75]:= Table[Print[i, " ", Factor[x^i + 64]], {i, 1, 12}]
```

1 $64+x$

2 $64+x^2$

3 $(4+x)\left(16-4x+x^2\right)$

4 $\left(8-4x+x^2\right)\left(8+4x+x^2\right)$

5 $64+x^5$

6 $\left(4+x^2\right)\left(16-4x^2+x^4\right)$

7 $64+x^7$

8 $\left(8-4x^2+x^4\right)\left(8+4x^2+x^4\right)$

9 $\left(4+x^3\right)\left(16-4x^3+x^6\right)$

10 $64+x^{10}$

11 $64+x^{11}$

12 $\left(2-2x+x^2\right)\left(2+2x+x^2\right)\left(4-4x+2x^2-2x^3+x^4\right)\left(4+4x+2x^2+2x^3+x^4\right)$

Example

Find all natural numbers *n* between 1 and 12 for which the polynomial $x^n + 64$ can be written as a product of two nonconstant polynomials with integer coefficients.

```
Table[Print[x^i + 64, " = ", Factor[x^i + 64]], {i, 1, 12}]
```

$64 + x = 64 + x$

$64 + x^2 = 64 + x^2$

$64 + x^3 = (4 + x)\left(16 - 4x + x^2\right)$

$64 + x^4 = \left(8 - 4x + x^2\right)\left(8 + 4x + x^2\right)$

$64 + x^5 = 64 + x^5$

$64 + x^6 = \left(4 + x^2\right)\left(16 - 4x^2 + x^4\right)$

$64 + x^7 = 64 + x^7$

$64 + x^8 = \left(8 - 4x^2 + x^4\right)\left(8 + 4x^2 + x^4\right)$

$64 + x^9 = \left(4 + x^3\right)\left(16 - 4x^3 + x^6\right)$

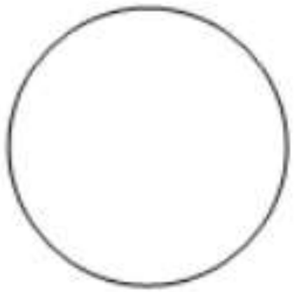
$64 + x^{10} = 64 + x^{10}$

$64 + x^{11} = 64 + x^{11}$

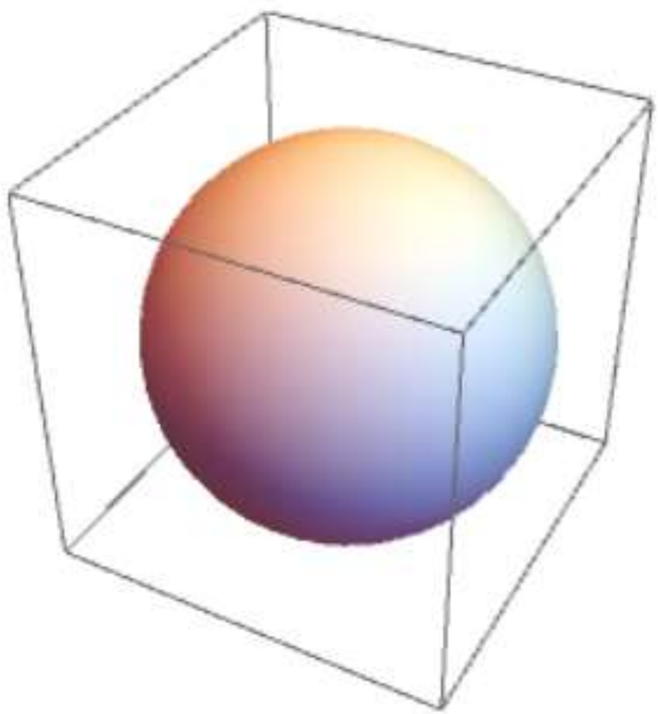
$64 + x^{12} = \left(2 - 2x + x^2\right)\left(2 + 2x + x^2\right)\left(4 - 4x + 2x^2 - 2x^3 + x^4\right)\left(4 + 4x + 2x^2 + 2x^3 + x^4\right)$

Print graphics:

```
In[1]:= Print[Graphics[Circle[]]]
```



```
In[2]:= Print[Graphics3D[Sphere[]]]
```



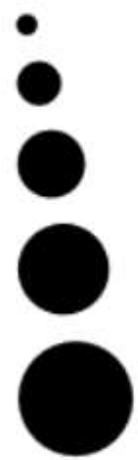
Print a column of expressions:

```
In[1]:= Print[Column[{x, aaa, z}]]

x
aaa
z
```

Print in a specified style:

```
In[1]:= Print[Style[aaa, 18, Red]]

aaa
```

```
Do[Print[Graphics[Disk[], ImageSize -> 10 n]], {n, 5}]
```



Logical Structures

Like other languages,
Mathematica supports the following logical structure:

Sequential:
Top to Bottom flow

Repetitive:
Loops: Do, While, For

Selective:
If true/false conditions

Clearing Values

Mathematica never forgets values assigned to a variable unless instructed to do so.

A common source of puzzling bugs is
the inadvertent reuse of previously defined variables or functions definitions.

Clear the value of a variable either before using it or immediately after using it.

To clear the value of the variable y, type

y = . or **Clear**[y].

Clear [y]

Several variables can be cleared together,

Clear[f, x, a]

To clear all items, use the following command:

Clear["Global`*"]

Do

Do[*expr*, *n*]
evaluates *expr* *n* times.

Do[*expr*, {*i*, i_{max}}]
evaluates *expr* with the variable *i* successively taking on the values 1 through i_{max} (in steps of 1).

Do[*expr*, {*i*, i_{min}, i_{max}}]
starts with $i = i_{min}$.

Do[*expr*, {*i*, i_{min}, i_{max}, *di*}]
uses steps *di*.

Do[*expr*, {*i*, {i_1, i_2, …}}]
uses the successive values i_1, i_2, ….

Do[*expr*, {*i*, i_{min}, i_{max}}, {*j*, j_{min}, j_{max}}, …]
evaluates *expr* looping over different values of *j* etc. for each *i*.

Print the first four squares:

```
In[1]:= Do[Print[n^2], {n, 4}]
1
4
9
16
```

Print 4 random integers:

```
In[1]:= Do[Print[RandomInteger[10]], 4]
3
10
1
3
```

n goes from −3 to 5 in steps of 2:

```
In[1]:= Do[Print[n], {n, -3, 5, 2}]
-3
-1
1
3
5
```

```
r = 0;
Do[If[EvenQ[i], Print[i, "  ", r]; Continue[]]; r += i, {i, 10}];
2  1
4  4
6  9
8  16
10  25
```

EvenQ

EvenQ[*expr*]
gives True if *expr* is an even integer, and False otherwise.

Test whether 8 is even:

```
In[1]:= EvenQ[8]
Out[1]= True
```

EvenQ gives False for non-numeric expressions:

```
In[1]:= EvenQ[x]
Out[1]= False
```

OddQ

OddQ[*expr*]
gives True if *expr* is an odd integer, and False otherwise.

Test whether 9 is odd:

```
In[1]:= OddQ[9]
Out[1]= True
```

OddQ gives False for non-numeric expressions:

```
In[1]:= OddQ[x]
Out[1]= False
```

Nested loops

In many applications there are several factors (variables) which change simultaneously, and this calls for what we call a *nested loop.* Instead of trying to describe the situation, let us look at some examples.

```
Do[Do [Print[i, " ", j], {j, 1, 2}],{i, 1, 3}]
```

Example: Find all the pairs (n, m) for $n, m \leq 10$ such that $n^2 + m^2$ is a square number (e.g., (3, 4) as $3^2 + 4^2 = 5^2$).

=⇒ Solution.

```
Do [Do [If[ Sqrt[i^2 + j^2] ∈ Integers, Print [i, " ", j]], {j, i, 10}], {i, 1, 10}]
```

Here is the result

3 4

6 8

One can make the nested Do loop a bit shorter. The following is an equivalent code to the first example of a Nested Do loop

```
Do[
 Print[i , " ", j],
 {i, 1, 3}, {j, 1, 2}]

1 1
1 2
2 1
2 2
3 1
3 2
```

Note that here `j` is the counter for the inner loop.

We have already seen the command `Table` which provides a sort of loop. In fact, `Table` can provide us with a nested loop as well.

```
Table[{i, j}, {i, 1, 3}, {j, 1, 2}]
{{{1, 1}, {1, 2}}, {{2, 1}, {2, 2}}, {{3, 1}, {3, 2}}}
```

While

While[*test*, *body*]
evaluates *test*, then *body*, repetitively, until *test* first fails to give True.

- While[*test*] does the loop with a null body.
- If Break[] is generated in the evaluation of *body*, the While loop exits.
- Continue[] exits the evaluation of *body*, and continues the loop.

Print and increment n while the condition $n < 4$ is satisfied:

```
In[1]:= n = 1; While[n < 4, Print[n]; n++]
1
2
3
```

```
n = 1; While[n++ < 4]; n
5
```

```
n = 1; While[n ≤ 5, Print[n^3]; n = n + 1]
1
8
27
64
125
```

The body can be included as part of the test:

```
In[1]:= n = 1; While[++n < 4]; n
Out[1]= 4
```

```
In[2]:= n = 1; While[n < 4, n++]; n
Out[2]= 4
```

For

For[*start*, *test*, *incr*, *body*]

executes *start*, then repeatedly evaluates *body* and *incr* until *test* fails to give True.

```
In[1]:= For[i = 0, i < 4, i++, Print[i]]
0
1
2
3
```

A comma delimits the parts of For; a semicolon delimits the parts of procedures:

```
In[1]:= For[i = 1; t = x, i^2 < 10, i++, t = t^2 + i; Print[t]]
```

$1 + x^2$

$2 + (1 + x^2)^2$

$3 + (2 + (1 + x^2)^2)^2$

```
In[91]:= For[t = 1; k = 1, k ≤ 5, k++, t *= k; Print[k, "  ", t]]
1  1
2  2
3  6
4  24
5  120
For[t = 1; k = 1, k ≤ 5, k++, t *= k; Print[k, "  ", t]; If[k < 2, Continue[]]; t += 2]
1  1
2  2
3  12
4  56
5  290
For[i = 1, i < 1000, i++; Print[i], If[i > 7, Break[]]];
2
3
4
5
6
7
8
```

Relational Expressions

MATHEMATICA has the following relational expressions:

Operator	Meaning
= =	Equal To
!=	Not Equal To
<	Less Than
>	Greater Than
<=	Less Than Or Equal To
>=	Greater Than Or Equal To

Two variables x and y in MATHEMATICA can be compared using the following relational statements:

(x = y)	true if x equals y otherwise false;
(x != y)	true if x and y are unequal otherwise false;
(x > y)	true if x is greater than y, false otherwise;
(x < y)	true if x is less than y, false otherwise;
(x >= y)	true if x is greater than or equal to y, false otherwise;
(x <= y)	true if x is less than or equal to y, false otherwise.

If

If[*condition*, *t*, *f*]
gives *t* if *condition* evaluates to True, and *f* if it evaluates to False.

If[*condition*, *t*, *f*, *u*]
gives *u* if *condition* evaluates to neither True nor False.

- If[*condition*, *t*] gives Null if *condition* evaluates to False.

```
x = 51; y = 65;
If[x == y, Print["x equals y"], Print["x is not equal to y"]]
```

```
x is not equal to y
```

If the condition is neither True nor False, If remains unevaluated:

```
In[1]:= If[a < b, 1, 0]
Out[1]= If[a < b, 1, 0]
```

The form with an explicit case for an undetermined condition evaluates in any case:

```
In[1]:= If[a < b, 1, 0, Indeterminate]

Out[1]= Indeterminate
```

Use TrueQ to force the condition to always return a Boolean value:

```
In[1]:= If[TrueQ[a < b], 1, 0]

Out[1]= 0
```

If can be used as a statement:

```
In[1]:= x = -2;
        If[x < 0, y = -x, y = x]; y

Out[1]= 2
```

It can also be used as an expression returning a value:

```
In[2]:= y = If[x < 0, -x, x]

Out[2]= 2
```

TrueQ

TrueQ[*expr*]
yields True if *expr* is True, and yields False otherwise.

- You can use TrueQ to "assume" that a test fails when its outcome is not clear.
- TrueQ[*expr*] is equivalent to If[*expr*, True, False, False].

TrueQ will return True only if the input is explicitly True:

```
In[1]:= TrueQ[True]
Out[1]= True

In[2]:= TrueQ[False]
Out[2]= False

In[3]:= TrueQ[x]
Out[3]= False
```

Logical operators

Relations given above may be combined with the following logical operator:

And, Or, Not

(A **&&** B) is true only if both A and B are true, otherwise it is false.

(A **||** B) is true if either A or B is true (both may be true), otherwise it is false.

(**!** A) is true if A is false, and false if A is true.

Example

```
x = 26
      If[ x <=50 && x >=10 ,
        Print["Given no. lies in [10, 50]"] ,
        Print["Given no. does not lie in [10, 50]"] ]
Given no. lies in [10, 50]
```

Transfer of Control: Unconditional Jumping

The simple Goto statement transfers
the control to another line within a procedure.

```
( ……
…….
…….
label ;
      …….
      …….
      …….
      If[ logical expression, Goto [ label ]]
      ……..
      ……..
      ……..
)
```

Goto

Goto[*tag*]

scans for Label[*tag*], and transfers control to that point.

Write a loop using Goto and Label:

```
In[1]:= f[a_] := Module[{x = 1., xp},
          Label[begin];
          If[Abs[xp - x] < 10^-8, Goto[end]];
          xp = x;
          x = (x + a / x) / 2;
          Goto[begin];
          Label[end];
          x]
```

```
In[2]:= f[2]
```

```
Out[2]= 1.41421
```

Find the sum of the sequence

$$\frac{1}{1+2}+\frac{2}{2+3}+\cdots+\frac{10}{10+11}.$$

$\Longrightarrow$ SOLUTION.

```
For[i = 1; sum = 0, i < 11, i++, sum += i/(i + i + 1)];
sum
64157087/14549535
```

In this example we build a function, using different styles, to calculate the sum of the square roots of n consecutive integers : $\{\sqrt{1}+...+\sqrt{n}\}$, and apply it with $n = 50$.

- A classical but inefficient way of doing this in *Mathematica*, is:

```
n = 50; sum = 0.0; Do[ sum = sum + N[Sqrt[i]], {i, 1, n} ];
sum

239.036
```

- In the functional style we transcribe almost literally the traditional notation in *Mathematica*. This approach is not only simpler but also more effective.

$$\texttt{rootsum[n\_] := } \sum_{i=1}^{n} \sqrt{i}$$

```
rootsum[50] // N
```

```
239.036
```

Special Characters: Mathematical and Other Notation

● — \[GrayCircle]

■ — \[GraySquare]

⋛ — \[GreaterEqualLess]

≥ — \[GreaterEqual]

≧ — \[GreaterFullEqual]

≫ — \[GreaterGreater]

≷ — \[GreaterLess]

⩾ — \[GreaterSlantEqual]

≳ — \[GreaterTilde]

ⅈ — \[ImaginaryI]

ⅉ — \[ImaginaryJ]

— \[ImplicitPlus]

⇒ — \[Implies]

— \[IndentingNewLine]

∞ — \[Infinity]

∫ — \[Integral]

∩ — \[Intersection]

ϵ — \[Epsilon]

== — \[Equal]

≂ — \[EqualTilde]

⇌ — \[Equilibrium]

⧦ — \[Equivalent]

⊠ — \[ErrorIndicator]

ESC — \[EscapeKey]

η — \[Eta]

ð — \[Eth]

€ — \[Euro]

∃ — \[Exists]

ⅇ — \[ExponentialE]

φ — \[FormalCurlyPhi]

δ — \[FormalDelta]

τ — \[FormalTau]

θ — \[FormalTheta]

∀ — \[ForAll]

a — \[FormalA]

α — \[FormalAlpha]

b — \[FormalB]

β — \[FormalBeta]

c — \[FormalC]

A — \[FormalCapitalA]

Α — \[FormalCapitalAlpha]

B — \[FormalCapitalB]

Β — \[FormalCapitalBeta]

C — \[FormalCapitalC]

Χ — \[FormalCapitalChi]

D — \[FormalCapitalD]

Δ — \[FormalCapitalDelta]

Φ — \[FormalCapitalPhi]

Π — \[FormalCapitalPi]

Ψ — \[FormalCapitalPsi]