Mathematica Notes

Cheatsheet

Information

- *?symbol* (Show information of that symbol)
- *??symbol* (Show more detailed information of that symbol)
- *?*symbol** (Show information of that symbol with regex. e.g. *?*Q*)

operator

- + (*Plus*), - (*Subtract*), * (*Times*), / (*Divide*), ^ (*Power*), += , -=, *=, /=
- > (*Greater*), != (*Unequal*), && (And), <= (*LessEqual*), || (Or), == (*Equal*, equal of value after evaluation)
- === (*SameQ,* equal of the whole expression), =!= (*UnsameQ*)

Results

- % (the last result)
- %% (n % means the output of last n runs), equals *%n* or *Out[n]*

Assignment

- *lhs* = *rhs* (*Set*, assign the evaluation result of rhs to lhs instantaneously)
- *lhs* := *rhs* (*SetDelayed*, assign the evaluation result of rhs to lhs whenever the lhs is utilized)

List

- *{a, b, c,{d,e}}* (A list is defined in {})
- *Level[list, {level_you_want_to_Part}]* (Return the list in the corresponding level. Level 0 means the whole list or the entire expression. Level 1 removes one layer from the top of the tree form.)
- *Position[list, element]* (Return the position of the element in a list.)
- *MapAt[function, list, position]* (Map the function on the specified position.)
- *MapThread* (Map on the columns of the matrix)
- *Thread* (e.g. *Thread[f[{a1, b1}, {b1,b2}, c]]* outputs *{f[a1, b1, c], f[b1,b2,c]}*. The *Thread* works on the column part of a list, and take the scalar as the public argument.)
- *Through* (e.g. *Through*[{f,g,h}, {a,b,c}] outputs {f[a,b,c], g[a,b,c], h[a,b,c]}. The *Through* make functions working on a single argument list.)

- *MapIndexed* (Some functions need take value + index as their arguments.)
- *Nest, NestList, FixedPoint*
- *TableForm[list], MatrixForm[list]*
- *[[ ]]* (*Part*)
- *list[[3]]* (*Part* the 3rd element)
- *list[[{3,2}]]* (*Part* the 3rd and 2nd elements)
- *list[[1,2]]* (*Part* the 2nd element in the 1st element)
- *Range[...] , Array[...], Table[expression, {variable, start, end, step}]* (The Table function is the most useful one)
- Only *AppendTo* and *PrependTo* will modify the list itself. All other list functions will not modify the original list.

String

The built-in string functions are similar with the built-in string functions. Like *StringLength*, *StringPosition*, *StringQ*, *LetterQ*, *DigitQ*, *StringMatchQ*, *StringTake*, *StringDrop*, *StringInsert* ...

```
In[◦]:= StringJoin["Hello ", "world", "!"]
In[◦]:= "Hello" <> " world" <> "!"
In[◦]:= a = 2; ToExpression["a+b"]
```

Symbol's property

A symbol in MMA can have 4 kinds of properties: ***Attributes****[sym]* (e.g. Protected, Listable)*, DefaultValues[sym], Options[sym]* (some rules attached to the sym)*, Messages[sym].*
The Options[] and Messages[] of a symbol do not affect the kernel's **evaluation** process, but the Attribute[] and DefaultValues[] do.
Kernel evaluation process: **1. evaluate atomic expressions to get normal expressions, 2. evaluate head, 3. evaluate downstream parts, 4. apply rules (Options[] work here)**.

Function (DownValues, UpValues)

MMA is fundamentally a functional programming language. Functional means that functions and data are indistinguishable: functions can be treated like data and passed between functions. MMA's kernel is also called the infinite evaluation system. The heuristic evaluation process can be regarded as *expression //. {all global rules}*. During the evaluation if function overloading occurred, e.g. *f[0]* vs *f[x_Integer]* (two UpValues of *f*), the MMA will always use the more specific rules.

- *@* (*Prefix*, e.g. *f @ x*, Prefix can only be used for functions with single argument. It is only for saving the [], with no additional semantics.)
- */@* (*Map*, e.g. *f /@ {a,b,c}* equals *{f[a], f[b], f[c]}*. It can also avoid the trouble of defining the *Listable* attribute for a symbol.)
- *@@* (*Apply*, e.g. *f @@ {a,b,c}* means *f[a,b,c].* The *Apply* means to replace the *Head* at level 0)
- *@@@* (MapApply, e.g. *f @@@ {{a,b},{c}}* means *{f[a, b], f[c]}*. *@@@* means replace *Head* at level 1)

- pure function. Like *Power @@ # &*. *#* (*Slot*) means formal variable and & terminates the pure function. We can also use *#1, #2, ...*
- // (*Postfix*, e.g. *x // f*)
- *x~f~y* (*Infix*)

Note that priority: *Prefix* > Infix > Postfix. No need to memorize. It is always a good habit to use ().

- *f[x_,y_] := expr* (_ is called *Blank*)
- *f[x_,y_] := first line; second line; ... last line* is called the compound expression.

Note that **modifying function arguments is not permitted in MMA**. This restriction exists because when functions are called, all arguments are evaluated and replaced wherever they appear within the function.

- ***Attributes****[]* show attributes of a function. e.g. *Protected*
- *Listable* (***Listable*** is an **attribute** of a function. Virtually all of the built-in numerical functions (e.g. *Plus, Sin, Gamma*), predictive functions (xxQ, e.g. *IntegerQ*, *PrimeQ*) and some of the symbolic functions (e.g. *Together, ToExpression*) have the *Listable* attribute. For a function *f* with *Listable* attribute, *f[{list}]* equals *{f[list[[1]]], f[list[[2]]] ...}*)
- *Clear, ClearAll, Remove* (*Clear* means to clear the Rules (*OwnValues*, *UpValues*, *DownValues* ...) attached with the symbol. *ClearAll* means to clear the the Rules, *Options*, and *Attributes* attached with the symbol. *Remove* means to remove the symbol from the symbol table (do not use *Remove*...Because its mechanism is not very clear.)
  e.g. `ClearAll /@ (($Context <> "*") // Names);`)
- Predication in MMA, like *Q functions, is used to predicate if an expression has some attributes.
- *Module* in MMA, is a function with local variables.
- Besides *Module*, the *With* function can also define local variables. The *name* in *With[{name=expr}, body]* is not a variable but a macro like *#define* in C; it will replace the *name* anywhere in *body* to *expr*, so that the *name* can not be assigned to another value after the first definition. Advantage: faster than *Module*.

Rules

The evaluation process of MMA is just like *expression //. {all global rules}*.

There is 6 types of rules . The symbol with rules is called the rule' s tag .

type of rule	for evaluation of	example
`OwnValues[sym]`	`sym`	`sym :=`
`DownValues[sym]`	`sym[...]`	`sym[...] :=`
`UpValues[sym]`	`head[... sym ...]` or `head[... _sym ...]`	`head[... sym ...] ^:=` or `sym /: head[... sym ...] :=`
`SubValues[sym]`	`sym[...][...]`	`sym[...][...] := ...`
`NValues[sym]`	`N[sym, precision]` or `N[sym[...], precision]`	`N[sym[...], precision] := ...`
`FormatValues[sym]`	`format[sym[...]]`	`Format[sym[...], format] :=`

A pattern is a collective term for a class of expressions, with the pattern itself is an expression.

- *a -> b* (form of a *Rule*)

- *a :> b* (*RuleDelayed*). **For *a->b*, b will be evaluated first before replacement. For *a:>b*, b will be replaced first and then be evaluated (every time when the rule is applied). Once the r.h.s b has a function form, just use :>.**

In[∘]:=
```
expr = Sqrt[u - 1] / Sqrt[u^2 - 1]
```
Out[∘]=

$$\frac{\sqrt{-1+u}}{\sqrt{-1+u^2}}$$

```
expr /. 1 / Sqrt[y_] → 1 / Sqrt[Factor@y]
  (*Factor @ y will be evaluated first before replacement by the rule,
  so that the rule equals to replace 1/Sqrt[y_] to 1/Sqrt[y],
  and thus no difference with the original form. *)
```
Out[∘]=

$$\frac{\sqrt{-1+u}}{\sqrt{-1+u^2}}$$

In[∘]:=
```
expr /. 1 / Sqrt[y_] :→ 1 / Sqrt[Factor@y]
```
Out[∘]=

$$\frac{\sqrt{-1+u}}{\sqrt{(-1+u)\ (1+u)}}$$

In[∘]:=
```
Clear[x, y]; {x, x_, y, y_} /. var_Pattern :→ q[var[[1]]]
```
Out[∘]=
```
{x, q[x], y, q[y]}
```

NOTE that, **both the a->b and a:>b will evaluate a first**!!! Unless using *HoldPattern[a]*

- /. (*ReplaceAll*[expr, rule])
- //. (*ReplaceRepeated*)
- ***Options****[symbol]* (check the options of a symbol (usually a function). **The options are shown in the rule format** and can usually be modified. The option rule is shown as name -> value and name :> value.)

In[∘]:=
```
Options[Plot]
```
In[∘]:=
```
Cases[{f[x], f[x, a → b], f[x, a :→ b, c → d], f[x, {a → b, c → d}, e → f]},
 f[arg1_, opt___?OptionQ] → {opt}]
```
In[∘]:=
```
f[x_, a → b] := x;
```
In[∘]:=
```
f[x, a → b]
```

- *SetOptions[symbol, option]*
- =. (*Unset* a rule. e.g. f[x_]=. means clear the *DownValues* of *f*, but the DownValues of *f[x_Integer]* will remain in *DownValues[f]*. *f* =. means to clear *OwnValues* of *f*.)
- *pattern:value*, pattern with default value. e.g. *x_:2*

In[∘]:=
```
Clear[f, a, b, c]
Cases[{f[x, y, z], f[x, y], f[x]}, f[a_, b_ : 2, c_ : 3] → {a, b, c}]
```
Out[∘]=
```
{{x, y, z}, {x, y, 3}, {x, 2, 3}}
```

- _. means a pattern with default value 0 and can input nothing. e.g. *f[x_, y_.]*
- __ (*BlankSequence.* One or more patterns. Different patterns are separated using ,)
- ___ (*BlankNullSequence*. Zero or more patterns. Different patterns are separated using ,)
- *_head* (pattern constraints. Only match those patterns whose Head is head)

```
factorial[n_Integer] := Product[i, {i, n}];
factorial[4]
factorial[3.5]
```

```
In[]:= squareAnyNumber[
          n_?IntegerQ | n_Real | n_Rational | n_Complex | (n_ /; VectorQ@n)] := n^2;
```

```
In[]:= squareAnyNumber /@ {3, 4.5, 2 / 3, 3 + 4 I, {1, 2, 3}, x}
```

- *_?testFunctionReturnTrueOrFalse* (pattern constraints. Only match those patterns that make test function(symbol) return True)

```
In[]:= Cases[{1, Sqrt[2], b}, _?IntegerQ]
```

- *pattern /; testFunctionReturnTrueOrFalse* (Condition. Only match those patterns that make test function return True)

```
In[]:= Cases[{-1, 0, 1, 10, 200, 5.0}, x_ /; 1 ≤ x ≤ 10 && IntegerQ[x]]
Out[]=
{1, 10}
```

- *name:pattern* (Assign the alias name to the pattern, similar to *n_*, with the advantage that the subsequent pattern can be sufficiently complex.)

```
In[]:= Cases[{{1, 2, 3}, a, {4, 5}}, t : {__Integer} → t^2]
Out[]=
{{1, 4, 9}, {16, 25}}
```

- *pattern..* (pattern repeated 1 or more times)
- *pattern...* (pattern repeated 0 or more times)

```
Cases[{{1, 2}, {3, a}, {b, 4, c}, {d, 2.5}}, {(_Integer | _Symbol) ...}]
(*The list only contains integer or symbol*)
Out[]=
{{1, 2}, {3, a}, {b, 4, c}}
```

- *pattern1* | *pattern2* (match pattern1 or pattern2)

```
In[]:= {Log[2], 4.5, 2 + 3 * I} /. x_Integer | x_Real → Sqrt[x]
Out[]=
```

$$\left\{\frac{\text{Log}[2]}{2}, 2.12132, 2+3\,i\right\}$$

- ^:= (UpSetDelayed, set *UpValues* of arguments in functions. When a symbol has a up valued rule, the rule will be checked upstream. i.e., the symbol is a part of the pattern and is not the head of the pattern. By contrast, a symbol attached with down valued rules itself is the head of the pattern.)

```
In[]:= Clear[log]
       Factor[log[x_ * y_]] ^:= log[x] + log[y];
```

In[•]:=
```
{DownValues@Factor, UpValues@Factor}
```
Out[•]=
```
{{}, {}}
```

In[•]:=
```
{DownValues@log, UpValues@log}
```
Out[•]=
```
{{}, {HoldPattern[Factor[log[x_ y_]]] :> log[x] + log[y]}}
```

In[•]:=
```
ClearAll@log
```
In[•]:=
```
log /: Factor[log[x_ * y_]] := log[x] * log[y];
```
In[•]:=
```
{DownValues@log, UpValues@log}
```
Out[•]=
```
{{}, {HoldPattern[Factor[log[x_ y_]]] :> log[x] log[y]}}
```

- /: ... := (*TagSetDelayed*. A better way to set *UpValues* of arguments in functions)
- To test patterns, we could use *MatchQ* (return True if an expression matched with a pattern) and *Cases* (return expressions satisfying a certain pattern)
- *Sequence* is the head for the expressions satisfying the form __ and ___. **The reason why Sequence head is rarely encountered is because whenever it is enclosed by another expression, it undergoes splicing** (referred to as sequence splicing):

Important concepts

Everything is an expression

In Mathematica, everything is an expression.
Every expression is a combination of normal expressions.
A normal expression can be constructed using 3 parts: 1. atoms (symbol, number, string), 2. square brackets [], 3. commas , .
The [] and commas can be regarded as the special symbols.

In summary, in MMA, everything is an expression, and every expression is a combination of symbols.

atoms

- **symbol**

The Symbol in MMA, is a sequence of letters, digits and character $.

All the system-defined atoms start with capital letters and $. (e.g. *Plus*, *D*, *FullForm*, *$Version*, *$MachinePrecision*)

The *, + , and / are still the symbols (*Times*, and *Plus*) in MMA.

Every expression has a **Head**, and the Head is called the **type** of the expression.

In[•]:=
```
Head[Plus]
```
Out[•]=
```
Symbol
```

In[∘]:=
```
Head[$MachinePrecision]
```
Out[∘]=
```
Real
```

In[∘]:=
```
Head /@ {3, 3 / 2, 3 + 2 * I, {3, 2}, a}
```
Out[∘]=
```
{Integer, Rational, Complex, List, Integer}
```

▪ number

The numbers in MMA have 4 types: *Integer*, *Real*, *Rational* (integer1 / integer2), *Complex* (a+b I)

In[∘]:=
```
Head[1.]
```
Out[∘]=
```
Real
```

In[∘]:=
```
Head[1 / 2]
```
Out[∘]=
```
Rational
```

In[∘]:=
```
Head[1 + 2 I]
```
Out[∘]=
```
Complex
```

Note that the **Head** of an expression is the top layer of its tree form,which is used for the evaluation process.

▪ string

The string in MMA is a sequence of characters enclosed by double quotes (“sth.”).

The \” stands for the single “.

In[∘]:=
```
Head["\"hello world\""]
```
Out[∘]=
```
String
```

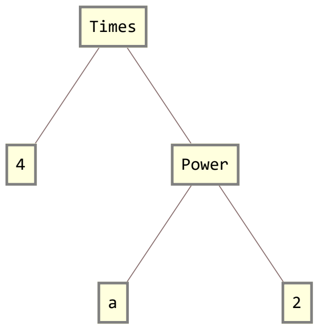
In[∘]:=
```
"\"hello world\""
```
Out[∘]=
```
"hello world"
```

Forms

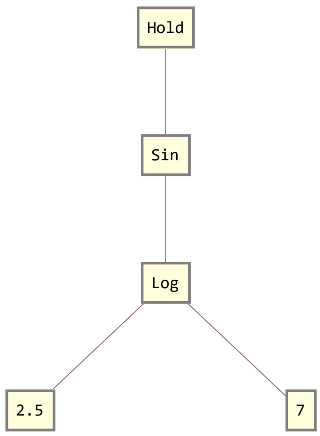
The *FullForm* and *TreeForm* expression show the expressions in MMA’s view. The evaluation process of a normal expression is from tree’s bottom to its top.

In[∘]:=
```
FullForm[4 * a^2]
```
Out[∘]//FullForm=
```
Times[4, Power[a, 2]]
```

In[∘]:=
```
TreeForm[4 * a^2]
```
Out[∘]//TreeForm=

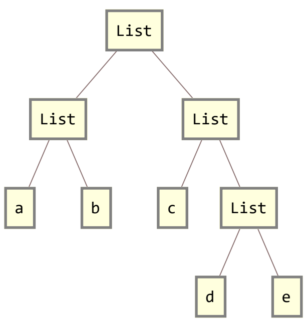


In[∘]:=
```
TreeForm[Hold[Sin[Log[2.5, 7]]]]
```
Out[∘]//TreeForm=



Another very import conception is level . level 0 means the entire expression, **level 1 removes one layer**, level 2 removes two layers and so on . We can use Position to get the position of a element in the list . If the Position function returns a list contains 2 integers like {1, 1}, it means the position in {level1, level2} .

In[∘]:=
```
alist = {{a, b}, {c, {d, e}}};
```
In[∘]:=
```
alist // TreeForm
```
Out[∘]//TreeForm=



In[∘]:=
```
Level[alist, {1}]
```
Out[∘]=
```
{{a, b}, {c, {d, e}}}
```

In[]:=
```
Level[alist, {2}]
```
Out[]=
```
{a, b, c, {d, e}}
```
In[]:=
```
Level[alist, {3}]
```
Out[]=
```
{d, e}
```
In[]:=
```
Position[alist, d]
```
Out[]=
```
{{2, 2, 1}}
```
In[]:=
```
MapAt[f, alist, {2, 2, 1}]
```
Out[]=
```
{{a, b}, {c, {f[d], e}}}
```

Evaluation process of an expression

The evaluation process in MMA involves the iterative replacement of rules associated with their corresponding symbols until no further rule replacement are possible.

Evaluation process:
-> write code in the front end
-> ***MakeExpression*** to convert your input from a textual or **box structure** into a Wolfram Language expression. This step translates the human-readable code into a form that the kernel can understand and process.
-> MMA kernel **evaluation**
-> ***MakeBoxes*** convert evaluated results into a **box structure** that can be displayed
-> Finally, the front end renders this box structure into a human-readable format, showing the result in the notebook or interface.

OwnValues

```
a := b + c
```
In[]:=
```
OwnValues[a]
```
Out[]=
```
{HoldPattern[a] :→ b + c}
```
In[]:=
```
Head[a]
```
Out[]=
```
Plus
```

DownValues

The user defined functions (in the form of *func[pattern_]*), in essence are to attach DownValues (one kind of rules) to a symbol *func*.

In[*]:=
```
ClearAll[f];
f[x_Integer] := x^2;
DownValues@f
```

Out[*]=
```
{HoldPattern[f[x_Integer]] :> x^2}
```

In[*]:=
```
f[x_Integer] =.
```

In[*]:=
```
DownValues@f
```

Out[*]=
```
{}
```

- A good example. Define a derivative function.

In[*]:=
```
ClearAll[diff]
```

In[*]:=
```
(*define diff as a linear operator*)
diff[c_ * f_ , x_] /; FreeQ[c, x] := c * diff[f, x];
diff[f_ + g_, x_] := diff[f, x] + diff[g, x];
```

In[*]:=
```
(*test diff is a linear operator*)
diff[3 x^2 - 2 x + 1, x]
```

Out[*]=
```
diff[1, x] - 2 diff[x, x] + 3 diff[x^2, x]
```

In[*]:=
```
(*add derivation rules*)
diff[c_, x_] /; FreeQ[c, x] := 0
diff[x_^n_ : 1, x_] /; FreeQ[n, x] := n * x^(n - 1)
```

In[*]:=
```
diff[3 * x^2 - 2 * x + 1, x]
```

Out[*]=
```
-2 + 6 x
```

In[*]:=
```
diff[(x + 1)^2, x]
```

Out[*]=
```
diff[(1 + x)^2, x]
```

In[*]:=
```
(*Implement chain rule*)
diff[fx_^n_, x_] /; FreeQ[n, x] && ! FreeQ[fx, x] := n * fx^(n - 1) * diff[fx, x]
```

In[*]:=
```
diff[(x + 1)^2, x]
```

Out[*]=
```
2 (1 + x)
```

In[*]:=
```
diff @@ {(x^2 + 2 * x + 1)^3, x}
```

Out[*]=
```
3 (2 + 2 x) (1 + 2 x + x^2)^2
```

In[•]:=
```
DownValues[diff]
```
Out[•]=
```
{HoldPattern[diff[c_ f_, x_] /; FreeQ[c, x]] :> c diff[f, x],
 HoldPattern[diff[f_ + g_, x_]] :> diff[f, x] + diff[g, x],
 HoldPattern[diff[c_, x_] /; FreeQ[c, x]] :> 0,
 HoldPattern[diff[x_^n_, x_] /; FreeQ[n, x]] :> n x^(n-1),
 HoldPattern[diff[x_^n_:1, x_] /; FreeQ[n, x]] :> n x^(n-1),
 HoldPattern[diff[fx_^n_, x_] /; FreeQ[n, x] && ! FreeQ[fx, x]] :> n fx^(n-1) diff[fx, x]}
```

UpValues

When a symbol has a up valued rule, the rule will be checked upstream. i.e., the symbol is a part of the pattern and is not the head of the pattern. By contrast, a symbol attached with down valued rules itself is the head of the pattern.

FormatValues

The rule as FormatValues controls the print form of the symbol.

In[•]:=
```
BesselJ[1, Pi / 2]
```
Out[•]=

BesselJ$\left[1, \frac{\pi}{2}\right]$

In[•]:=
```
Unprotect[BesselJ];
Format[BesselJ[n_, x_]] := Subscript[J, n][x]
Protect[BesselJ];
BesselJ[1, Pi / 2]
```
Out[•]=

$J_1\left[\frac{\pi}{2}\right]$

In[•]:=
```
(* It can also be done in reverse *)
Unprotect[Subscript];

Subscript[J_, n_][x_] := BesselJ[n, x] /; J === J

Protect[Subscript];
```

In[•]:=
```
FormatValues[BesselJ]
```
Out[•]=
```
{HoldPattern[MakeBoxes[J_n_[x_], FormatType_]] :> Format[J_n[x], FormatType],
 HoldPattern[J_n_[x_]] :> J_n[x]}
```

SubValues

The rule as SubValues which has the form *sym[...][...]* is very rare in MMA.
Do not use SubValues.

Hold Family

- 4 Hold- attributes

- ***HoldFirst***. Prevent evaluation of the first argument of a function.
- ***HoldRest***. Prevent evaluation of the all but first arguments of a function.
- **HoldAll**. Prevent evaluation of the all arguments of a function. e.g. *SetAttributes[func, HoldAll]*
- ***HoldAllComplete****.* Prevent any modification or evaluation of function's arguments, including *Sequence* and *Evaluate*.

```
In[•]:= SetAttributes[f, HoldAll]
In[•]:= f[x_] := g[x]
In[•]:= Trace[f[2 + 2]]
Out[•]= {f[2 + 2], g[2 + 2], {2 + 2, 4}, g[4]}
```

- 6 Hold- functions
  - ***Hold***
  - ***HoldForm***. Mainly used by *Trace* function. Unlike *Hold*, which is visible in the output, *HoldForm* does not show itself in the output, making it look like the expression hasn't been evaluated, but without showing the *Hold* wrapper.
  - ***HoldComplete****.* It prevents not only the evaluation of its arguments but also the evaluation of any part of its arguments. It blocks even certain types of simplifications or evaluations that Hold might allow. e.g. *Hold[Evaluate[1 + 1]]* outputs 2, while *HoldComplete[Evaluate[1 + 1]]* outputs *HoldComplete[Evaluate[1 + 1]]*
  - ***Unevaluated****.* Unevaluated is very similar with Hold, but is **invisible to the outer function**. The opposite side is *Evaluate* function. *Unevaluated* is used to temporarily prevent the evaluation of an expression. It wraps an expression in a way that delays evaluation for one step in the evaluation sequence. After that step, the expression may be evaluated unless it's within another holding construct (decided by the outer wrapp function).
  - ***HoldPattern***. *HoldPattern* is a function with *HoldAll* attribute, but it only works on pattern. *HoldPattern* is only necessary when patterns in *Hold*.

```
In[•]:= Hold[3 * 3] /. x_ * x_ → x^2
Out[•]= Hold[3 × 3]
In[•]:= Clear[x]; Hold[3 * 3] /. HoldPattern[x_ * x_] → x^2
Out[•]= Hold[3^2]
In[•]:= ClearAll[f]; f[x_] := x^2; DownValues[f]
Out[•]= {HoldPattern[f[x_]] :→ x^2}
```

  - *Verbatim*. Verbatim specifies that the expression it contains must be matched exactly, w/o any special interpretation by the pattern matcher.

```
In[•]:= Clear[x, y]; {x, x_, y, y_} /. Verbatim[x_] → matched
Out[•]= {x, matched, y, y_}
```

Note that *Hold* and *HoldForm* are just functions have *HoldAll* attribute. The only difference between *Hold* and *HoldForm*, is that in standard output format, the head *HoldForm* does not appear.

```
In[●]:= {Hold[1 + 1], HoldForm[1 + 1]}
Out[●]=
        {Hold[1 + 1], 1 + 1}

In[●]:= InputForm[%]
Out[●]//InputForm=
        {Hold[1 + 1], HoldForm[1 + 1]}

In[●]:= Head[HoldForm[x = 7]]
Out[●]=
        HoldForm
```