Fast Introduction For Programmers

Basics

- Function arguments use square brackets [...]
- Everything in the Wolfram Language is a symbolic expression: head[arguments]
- Fully symbolic, so "undefined variables" can always just stand for themselves
- Lists are indicated by {...}, indexing starts at 1, index using [[..]], "spans" in lists using ;;
- The standard "iterator specification": min, max, step
- = immediate assignment, := delayed, =. clear
- start variable names with lowercase letters, reserving capitals for built-in objects:

In[14]:=
```
Table[FromCharacterCode[127 876], {i, 5}, {j, i}] // MatrixForm(* outer index first *)
```

Out[14]//MatrixForm=

$$\begin{pmatrix} \{🎄\} \\ \{🎄, 🎄\} \\ \{🎄, 🎄, 🎄\} \\ \{🎄, 🎄, 🎄, 🎄\} \\ \{🎄, 🎄, 🎄, 🎄, 🎄\} \end{pmatrix}$$

Patterns

- Patterns stand for classes of expressions. The basic pattern construct _ (pronounced "blank") stands for any expression.
- /. means "replace everywhere"
- __ ("double blank") stands for any sequence of expressions
- a | b | c stands for a, b or c
- _h stands for any expression with head h
- :> is a delayed rule—the analog of := for a rule

In[19]:=
```
{f[1], f[5], f[x]} /. f[x_] → x + 4
```

Out[19]= {5, 9, 4 + x}

In[20]:=
```
{f[1], f[5], f[x]} /. x_ → x + 4
```

Out[20]= {4 + f[1], 4 + f[5], 4 + f[x]}

In[23]:=
```
{f[1], f[5], f[x]} /. f[x_] → g[x + 4]
```
Out[23]=
```
{g[5], g[9], g[4 + x]}
```

Functions

- function definitions are just assignments that give transformation rules for patterns
- /; to restrict a definition to apply only when a particular condition holds
- pure functions, indicated by ending with & Their first argument is indicated by # (anonymous functions, lambda expressions
- /@ ("slash at") is a short notation for Map, Map can operate at specific levels.
- Apply applies a function to multiple arguments, @@ is equivalent to Apply, operating by default at level 0, @@@ means "apply at level 1, @ means ordinary function application
- Many built-in functions in the Wolfram Language can use "functional" or "operator" forms.
- Options gives the default options of a function, option:>value to make value be reevaluated every time the option is used; When pure functions are given as options, they need to be put in parentheses

In[24]:=
```
Map[f, {a, b, c, d}]
```
Out[24]=
```
{f[a], f[b], f[c], f[d]}
```

In[25]:=
```
Apply[f, {a, b, c, d}]
```
Out[25]=
```
f[a, b, c, d]
```

In[27]:=
```
f @@ {a, b, c, d}
```
Out[27]=
```
f[a, b, c, d]
```

In[29]:=
```
f @@@ {{a}, {b}, {c}, {d}}
```
Out[29]=
```
{f[a], f[b], f[c], f[d]}
```

In[30]:=
```
f @@@ {a, b, c, d}
```
Out[30]=
```
{a, b, c, d}
```

In[31]:=
```
f@{a, b, c, d}
```
Out[31]=
```
f[{a, b, c, d}]
```

Procedures

- Procedural programming is usually needed only in small doses
- Use ; to separate different operations
- Module does lexical scoping (localizing names), Block does dynamic scoping (localizing values), DynamicModule does scoping within a document

- Every time a module is evaluated, a new temporary symbol is created, and are removed if they are no longer referenced; Block localizes values only; it does not create new symbols
- If there is no need to assign to a local variable, a constant should be used instead
- Sow/Reap and Throw/Catch are useful ways to transfer data and control in procedural programs

In[32]:=
```
x = 7; Module[{x = x}, x = x + 1; x]
```
Out[32]= 8

In[33]:=
```
x
```
Out[33]= 7

In[39]:=
```
x = 7; Block[{}, x = x + 1; x]
```
Out[39]= 8

In[40]:=
```
x
```
Out[40]= 8

In[41]:=
```
Sum[i ^ 2 + 1, {i, 10}]
```
Out[41]= 395

In[42]:=
```
Reap[Sum[Sow[i ^ 2] + 1, {i, 10}]]
```
Out[42]= {395, {{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}}}

Numbers & Strings

- by default does exact computation whenever it can, N to get (potentially faster) numerical results
- ` to explicitly indicate the precision to assume in a number, I represents the imaginary unit
- Strings are indicated by double quotes ("..."), <> joins string
- any Unicode characters, entered using names like α, shortcuts like ESC A ESC, explicit Unicode like \:number α—or entered from a palette button
- ~~ to combine strings with pattern constructs, p... means zero or more repetitions of p,
- String templates use `` to indicate "slots" and <* ... *> to indicate expressions to evaluate

In[44]:=
```
ToCharacterCode["🤯"]
```
Out[44]= {127 877}

In[45]:=
```
FromCharacterCode[127 877]
```
Out[45]= 🤯

In[50]:=
```
FromCharacterCode[16^^1f385]
```
Out[50]= 🤯

Associations

- Associations associate keys and values (OrderedDict)
- a pure function, #key picks out the value corresponding to "key" in an association
- can mix associations and lists, and pick out parts using [[...]]

In[51]:=
```
d = <|"a" → "aaa", "b" → "bbb"|>
```
Out[51]=
```
<|a → aaa, b → bbb|>
```

In[52]:=
```
d["b"]
```
Out[52]=
```
bbb
```

In[53]:=
```
TemplateApply["first `a`, second `b`", d]
```
Out[53]=
```
first aaa, second bbb
```

Making Documents

cmd/alt + 1: Title, + 4 Section, +5 Subsection, +7 Text, + 8 code

External Connections

Type > at the beginning of the line

In[55]:=
```
> [i/2 for i in range(10)]
```

Syntax: Incomplete expression; more input is needed .

In[54]:=
```
ExternalEvaluate[{"Python", "Version" → "3.6"}, "[i/2 for in range(10)]"]
```
ExternalEvaluate: ExternalEvaluate is not supported in the Wolfram Cloud.

Out[54]=
```
$Failed
```