Hyperpolyglot

Computer Algebra I: Mathematica, SymPy, Sage, Maxima

a side-by-side reference sheet

sheet one: grammar and invocation | variables and expressions | arithmetic and logic | strings | arrays | sets | arithmetic sequences | dictionaries | functions | execution control | exceptions | streams | files | directories | libraries and namespaces | reflection

sheet two: symbolic expressions | calculus | equations and unknowns | optimization | vectors | matrices | combinatorics | number theory | polynomials | trigonometry | special functions | permutations | descriptive statistics | distributions | statistical tests

bar charts | scatter plots | line charts | surface charts | chart options

	mathematica	sympy	sage	maxima
version used	*10.0*	*Python 2.7; SymPy 0.7*	*6.10*	*5.37*
show version	*select* About Mathematica *in* Mathematica *menu*	sympy.__version__	$ sage --version *also displayed on worksheet*	$ maxima --version

implicit prologue		from sympy import *
		# enable LaTeX rendering in Jupyter notebook:
		init_printing()
		# unknown variables must be declared:
		x, y = symbols('x y')
			# unknowns other than x must be declared:
			y = var('y')	

grammar and invocation				
	mathematica	**sympy**	**sage**	**maxima**

interpreter	$ cat > hello.m
	Print["Hello, World!"]
	$ MathKernel -script hello.m
		if foo.py *imports sympy:*
		$ python *foo*.py
			$ cat > hello.sage
			print("Hello, World!")
			$ sage hello.sage
				$ cat >> hello.max
				print("Hello, world!");
				$ maxima -b hello.maxima

repl	$ MathKernel	$ python >>> from sympy import *	$ sage	$ maxima
block delimiters	(*stmt*; *…*)	: *and offside rule*	: *and offside rule*	block([x: 3, y: 4], x + y); /* Multiple stmts are separated by commas; a list of assignments can be used to set variables local to the block. */
statement separator	; *or sometimes newline* *A semicolon suppresses echoing value of previous expression.*	*newline or* ; *newlines not separators inside (), [], {}, triple quote literals, or after backslash: \*	*newline or* ; *newlines not separators inside (), [], {}, triple quote literals, or after backslash: \*	; *or* $ *The dollar sign $ suppresses output.*
end-of-line comment	*none*	1 + 1 # addition	1 + 1 # addition	*none*
multiple line comment	1 + (* addition *) 1	*none*	*none*	1 + /* addition */ 1;
variables and expressions				
	mathematica	**sympy**	**sage**	**maxima**
assignment	a = 3 Set[a, 3] (* rhs evaluated each time a is accessed: *) a := x + 3 SetDelayed[a, x + 3]	a = 3	a = 3	a: 3;
parallel assignment	{a, b} = {3, 4} Set[{a, b}, {3, 4}]	a, b = 3, 4	a, b = 3, 4	[a, b]: [3, 4]
compound assignment	+= -= *= /= *corresponding functions:* AddTo SubtractFrom TimeBy DivideBy	+= -= *= /= //= %= ^= **=	+= -= *= /= //= %= **=	*none*
increment and decrement	++x --x PreIncrement[x] PreDecrement[x] x++ x-- Increment[x] Decrement[x]	*none*	*none*	*none*
non-referential identifier	*any unassigned identifier is non-referential*	x, y, z, w = symbols('x y z w')	y, z, w = var('y z w') # x is non-referential unless assigned a value	*any unassigned identifier is non-referential*

identifier as value	x = 3
	y = HoldForm[x]
				x: 3;
				y: 'x;

global variable	*variables are global by default*	g1, g2 = 7, 8
		
		def swap_globals():
		  global g1, g2
		  g1, g2 = g2, g1
			g1, g2 = 7, 8
			
			def swap_globals():
			  global g1, g2
			  g1, g2 = g2, g1
				variables are global by default

local variable	Module[{x = 3, y = 4}, Print[x + y]] (* makes x and y read-only: *) With[{x = 3, y = 4}, Print[x + y]] (* Block[ ] declares dynamic scope *)	*assignments inside functions are to local variables by default*	*assignments inside functions are to local variables by default*	block([x: 3, y: 4], print(x + y));
null	Null	None	None	*no null value*
null test	x == Null	x is None	x is None	*no null value*
undefined variable access	*treated as an unknown number*	*raises* NameError	*raises* NameError	*treated as an unknown number*
remove variable binding	Clear[x] Remove[x]	del x	del x	kill(x);
conditional expression	If[x > 0, x, -x]	x if x > 0 else -x	x if x > 0 else -x	if x < 0 then -x else x;
arithmetic and logic				
	mathematica	**sympy**	**sage**	**maxima**
true and false	True False	True False	True False	true false
falsehoods	False	False 0 0.0	False None 0 0.0 '' [] {}	
logical operators	! True || (True && False) Or[Not[True], And[True, False]]	Or(Not(True), And(True, False)) # when arguments are symbols: ~ x | (y & z)		and or not
relational expression	1 < 2			is(1 < 2);
relational operators	== != > < >= <= *corresponding functions:* Equal Unequal Greater Less GreaterEqual LessEqual	Eq Ne Gt Lt Ge Le # when arguments are symbols: == != > < >= <=	== != > < >= <=	= # > < >= <=
arithmetic operators	+ - * / Quotient Mod *adjacent terms are multiplied, so * is not necessary.* Quotient *and* Mod *are functions, not binary infix operators. These functions are also available:* Plus Subtract Times Divide	+ - * / *??* % *if an expression contains a symbol, then the above operators are rewritten using the following classes:* Add Mul Pow Mod	+ - * / // %	+ - * / quotitent() mod() quotient *and* mod *are functions, not binary infix operators.*
integer division	Quotient[a, b]		7 // 3	quotient(7, 3);
integer division by zero	*dividend is zero:* Indeterminate *otherwise:* ComplexInfinity		*raises* ZeroDivisionError	*error*
float division	*exact division:* a / b			a / b
float division by zero	*dividend is zero:* Indeterminate *otherwise:* ComplexInfinity			*error*

power	2 ^ 32
	Power[2, 32]
		2 ** 32
		Pow(2, 32)
			2 ^ 32
			2 ** 32
				2 ^ 32;
				2 ** 32;

sqrt	*returns symbolic expression:* Sqrt[2]	sqrt(2)	sqrt(2)	sqrt(2);
sqrt -1	I	I	I	%i
transcendental functions	Exp Log Sin Cos Tan ArcSin ArcCos ArcTan ArcTan ArcTan *accepts 1 or 2 arguments*	exp log sin cos tan asin acos atan atan2	exp log sin cos tan asin acos atan atan2	exp log sin cos tan asin acos atan atan2
transcendental constants *π and Euler's number*	Pi E EulerGamma	pi E	pi e euler_gamma	%pi %e %gamma

float truncation *round towards zero, round to nearest integer, round down, round up*	IntegerPart Round Floor Ceiling	floor ceiling	int round floor ceil	truncate round floor ceiling
absolute value *and signum*	Abs Sign	Abs sign	abs sign	abs sign sign *returns* pos, neg, *or* zero
integer overflow	*none, has arbitrary length integer type*	*none, has arbitrary length integer type*	*none, has arbitrary length integer type*	*none, has arbitrary length integer type*
float overflow	*none*			*none*

rational construction	2 / 7	Mul(2, Pow(7, -1))
		Rational(2, 7)
			2 / 7	2 / 7

rational decomposition	Numerator[2 / 7]
	Denominator[2 / 7]
		numer, denom = fraction(Rational(2, 7))	numerator(2 / 7)
			denominator(2 / 7)
				num(2 / 7);
				denom(2 / 7);

decimal approximation	N[2 / 7]
	2 / 7 + 0.
	2 / 7 // N
	N[2 / 7, 100]
		N(Rational(2, 7))
		N(Rational(2, 7), 100)
			n(2 / 7)
			n(2 / 7, 100)
			
			# synonyms for n:
			N(2 / 7)
			numerical_approx(2 / 7)
				2 / 7, numer;

complex construction	1 + 3I	1 + 3 * I	1 + 3 * I	1 + 3 * %i;
complex decomposition *real and imaginary part, argument and modulus, conjugate*	Re Im Arg Abs Conjugate	re im Abs arg conjugate	(3 + I).real() (3 + I).imag() abs(3 + I) arg(3 + I) (3 + I).conjugate()	realpart imagpart cabs carg conjugate

random number
uniform integer, uniform float
	RandomInteger[{0, 99}]
	RandomReal[]
				random(100);
				random(1.0);

random seed *set, get*	SeedRandom[17] *??*			set_random_state(make_random_state(17)); *??*
bit operators	BitAnd[5, 1] BitOr[5, 1] BitXor[5, 1] BitNot[5] BitShiftLeft[5, 1] BitShiftRight[5, 1]			*none*

binary, octal, and hex literals	2^^101010
	8^^52
	16^^2a
				ibase: 2;
				101010;
				
				ibase: 8;
				52;
				
				/* If first hex digit is a letter, prefix a zero: */
				ibase: 16;
				2a;

radix	BaseForm[42, 7]
	BaseForm[7^^60, 10]
				obase: 7;
				42;

to array of digits	(* base 10: *) IntegerDigits[1234] (* base 2: *) IntegerDigits[1234, 2]			
		strings		
	mathematica	**sympy**	**sage**	**maxima**
string literal	"don't say \"no\""	*use* Python strings	*use* Python strings	"don't say \"no\""
newline in literal	*yes*			*Newlines are inserted into strings by continuing the string on the next line. However, if the last character on a line inside a string is a backslash, the backslash and the following newline are omitted.*
literal escapes	\\ \" \b \f \n \r \t \ooo			\" \\
concatenate	"one " <> "two " <> "three"			concat("one ", "two ", "three");

translate case	ToUpperCase["foo"]
	ToLowerCase["FOO"]
				supcase("foo");
				sdowncase("FOO");

trim	StringTrim[" foo "]			strim(" ", " foo ");
number to string	"value: " <> ToString[8]			concat("value: ", 8);

string to number	7 + ToExpression["12"]
	73.9 + ToExpression[".037"]
				7 + parse_string("12");
				73.9 + parse_string(".037");
				
				/* parse_string raises error if the string does not contain valid Maxima code. Use numberp predicate to verify that the return value is numeric. */

string join	StringJoin[Riffle[{"foo", "bar", "baz"}, ","]]			simplode(["foo", "bar", "baz"], ",");
split	StringSplit["foo,bar,baz", ","]			split("foo,bar,baz", ",");
substitute *first occurrence, all occurences*	s = "do re mi mi" re = RegularExpression["mi"] StringReplace[s, re -> "ma", 1] StringReplace[s, re -> "ma"]			ssubst("mi", "ma", "do re mi mi mi"); ssubstfirst("mi", "ma", "do re mi mi mi");
length	StringLength["hello"]			slength("hello");
index of substring	StringPosition["hello", "el"][[1]][[1]] (* The index of the first character is 1.*) (* StringPosition returns an array of pairs, one for each occurrence of the substring. Each pair contains the index of the first and last character of the occurrence. *)			ssearch("el", "hello"); /* 1 is index of first character; returns false if substring not found */ _
extract substring	(* "el": *) StringTake["hello", {2, 3}]			substring("hello", 2, 4);
character literal	*none*			*none*
character lookup	Characters["hello"][[1]]			

chr and ord	FromCharacterCode[{65}]
	ToCharacterCode["A"][[1]]
				ascii(65);
				cint("A");

delete characters	rules = {"a" -> "", "e" -> "", "i" -> "",
	  "o" -> "", "u" -> ""}
	StringReplace["disemvowel me", rules]			

		arrays		
	mathematica	**sympy**	**sage**	**maxima**
literal	{1, 2, 3} List[1, 2, 3]	*use* Python lists	*use* Python lists	[1, 2, 3];
size	Length[{1, 2, 3}]			length([1, 2, 3]);

lookup	(* access time is O(1) *)
	(* indices start at one: *)
	{1, 2, 3}[[1]]
	
	Part[{1, 2, 3}, 1]
				a: [6, 7, 8];
				a[1];

update	a[[1]] = 7			a[1]: 7;
out-of-bounds behavior	*left as unevaluated* Part[] *expression*			*Error for both lookup and update.*

element index	(* Position returns list of all positions: *)
	First /@ Position[{7, 8, 9, 9}, 9]
				a: [7, 8, 9, 9];
				first(sublist_indices(a, lambda([x], x = 9)));

slice	{1, 2, 3}[[1 ;; 2]]			
array of integers as index	(* evaluates to {7, 9, 9} *) {7, 8, 9}[[{1, 3, 3}]]			

manipulate back	a = {6,7,8}
	AppendTo[a, 9]
	elem = a[[Length[a]]]
	a = Delete[a, Length[a]]
	elem			

manipulate front	a = {6, 7, 8}
	PrependTo[a, 5]
				a: [6, 7, 8];
				push(5, a);

	`elem = a[[1]]`
	`a = Delete[a, 1]`
	`elem`
				`elem: pop(a);`

head	`First[{1, 2, 3}]`			`first([1, 2, 3]);`
tail	`Rest[{1, 2, 3}]`			`rest([1, 2, 3]);`

cons	`(* first arg must be an array *)`
	`Prepend[{2, 3}, 1]`
				`cons(1, [2, 3]);`

concatenate	`Join[{1, 2, 3}, {4, 5, 6}]`			`append([1, 2, 3], [4, 5, 6]);`
replicate	`tenZeros = Table[0, {i, 0, 9}]`			`ten_zeros: makelist(0, 10);`
copy	`a2 = a`			`a2: copylist(a);`
iterate	`Do[Print[i], {i, {1, 2, 3}}]`			`for i in [1, 2, 3] do print(i);`
reverse	`Reverse[{1, 2, 3}]`			`reverse([1, 2, 3]);`

sort	`(* original list not modified: *)`
	`a = Sort[{3, 1, 4, 2}]`
				`sort([3, 1, 4, 2]);`

dedupe	`DeleteDuplicates[{1, 2, 2, 3}]`			`unique([1, 2, 2, 3]);`

membership	`MemberQ[{1, 2, 3}, 2]`			`member(7, {1, 2, 3});`
				`evalb(7 in {1, 2, 3});`

map	`Map[Function[x, x x], {1, 2, 3}]`
	
	`Function[x, x x] /@ {1, 2, 3}`
	
	`(* if function has Listable attribute, Map is unnecessary: *)`
	`sqr[x_] := x * x`
	`SetAttributes[sqr, Listable]`
	`sqr[{1, 2, 3, 4}]`
				`map(lambda([x], x * x), [1, 2, 3]);`

filter	`Select[{1, 2, 3}, # > 2 &]`			`sublist([1, 2, 3], lambda([x], x > 2));`
reduce	`Fold[Plus, 0, {1, 2, 3}]`			
universal and existential tests	*none*			

min and max element	`Min[{6, 7, 8}]`
	`Max[{6, 7, 8}]`
				`apply(min, [6, 7, 8]);`
				`apply(max, [6, 7, 8]);`

shuffle and sample	`x = {3, 7, 5, 12, 19, 8, 4}`
	
	`RandomSample[x]`
	`RandomSample[x, 3]`			

flatten
one level, completely
	`Flatten[{1, {2, {3, 4}}}, 1]`
	`Flatten[{1, {2, {3, 4}}}]`
				`/* completely: */`
				`flatten([1, [2, [3, 4]]]);`

zip	`(* list of six elements: *)`
	`Riffle[{1, 2, 3}, {"a", "b", "c"}]`
	
	`(* list of lists with two elements: *)`
	`Inner[List, {1, 2, 3}, {"a", "b", "c"}, List]`
	
	`(* same as Dot[{1, 2, 3}, {2, 3, 4}]: *)`
	`Inner[Times, {1, 2, 3}, {2, 3, 4}, Plus]`
				`/* list of six elements: */`
				`join([1, 2, 3], ["a", "b", "c"]);`

cartesian product	`Outer[List, {1, 2, 3}, {"a", "b", "c"}]`			

sets

	mathematica	sympy	sage	maxima

literal	`(* same as arrays: *)`
	`{1, 2, 3}`
		`{1, 2, 3}`	`{1, 2, 3}`	`{1, 2, 3}`

size	`Length[{1, 2, 3}]`	`len({1, 2, 3})`	`len({1, 2, 3})`	`cardinality({1, 2, 3});`
array to set	`DeleteDuplicates[{1, 2, 2, 3}]`	`set([1, 2, 3])`	`set([1, 2, 3])`	`setify([1, 2, 3]);`
set to array	*none; sets are arrays*	`list({1, 2, 3})`	`list({1, 2, 3})`	`listify({1, 2, 3});`
membership test	`MemberQ[{1, 2, 3}, 7]`	`7 in {1, 2, 3}`	`7 in {1, 2, 3}`	`elementp(7, {1, 2, 3});`

subset test	`SubsetQ[{1, 2, 3}, {1, 2}]`	`{1, 2} <= {1, 2, 3}`
		`{1, 2}.issubset({1, 2, 3})`
			`{1, 2} <= {1, 2, 3}`
			`{1, 2}.issubset({1, 2, 3})`
				`subsetp({1, 2}, {1, 2, 3});`

		{1, 2, 3} >= {1, 2} {1, 2, 3}.issuperset({1, 2})	{1, 2, 3} >= {1, 2} {1, 2, 3}.issuperset({1, 2})	

universal and existential tests				every(lambda([x], x > 2), [1, 2, 3]);
				
				some(lambda([x], x > 2), [1, 2, 3]);

union	Union[{1, 2}, {2, 3, 4}]	{1, 2, 3} | {2, 3, 4} {1, 2, 3}.union({2, 3, 4})	{1, 2, 3} | {2, 3, 4} {1, 2, 3}.union({2, 3, 4})	union({1, 2, 3}, {2, 3, 4});
intersection	Intersect[{1, 2}, {2, 3, 4}]	{1, 2, 3} & {2, 3, 4} {1, 2, 3}.intersection({2, 3, 4})	{1, 2, 3} & {2, 3, 4} {1, 2, 3}.intersection({2, 3, 4})	intersection({1, 2, 3}, {2, 3, 4});
relative complement	Complement[{1, 2, 3}, {2}]	{1, 2, 3} - {2, 3, 4} {1, 2, 3}.difference({2, 3, 4})	{1, 2, 3} - {2, 3, 4} {1, 2, 3}.difference({2, 3, 4})	setdifference({1, 2, 3}, {2, 3, 4});
powerset			set(Set({1, 2, 3}).subsets())	powerset({1, 2, 3});
cartesian product	Outer[List, {1, 2, 3}, {"a", "b", "c"}]			cartesian_product({1, 2, 3}, {"a", "b", "c"});

arithmetic sequences

	mathematica	sympy	sage	maxima
unit difference	Range[1, 100]	range(1, 101)	range(1, 101)	makelist(i, i, 1, 100);
difference of 10	Range[1, 100, 10]	range(1, 100, 10)	range(1, 100, 10)	makelist(i, i, 1, 100, 10);
difference of 1/10	Range[1, 100, 1/10]	[1 + Rational(1,10)*i for i in range(0, 991)]	[1 + (1/10)*i for i in range(0, 991)]	makelist(i, i, 1, 100, 1/10);

dictionaries

	mathematica	sympy	sage	maxima
literal	d = <|"t" -> 1, "f" -> 0|> (* or convert list of rules: *) d = Association[{"t" -> 1, "f" -> 0}] (* and back to list of rules: *) Normal[d]	*use* Python dictionaries	*use* Python dictionaries	d: [["t", 1], ["f", 0]];
size	Length[Keys[d]]			length(d);
lookup	d["t"]			assoc("t", d);
update	d["f"] = -1			d2: cons(["f", -1], sublist(d, lambda([p], p[1] # "f")));
missing key behavior	*Returns a symbolic expression with head "Missing". If the lookup key was "x", the expression is:* Missing["KeyAbsent", "x"]			assoc *returns* false
is key present	KeyExistsQ[d, "t"]			
iterate				

keys and values as arrays	Keys[d]
	Values[d]
				map(lambda([p], p[1]), d);
				map(lambda([p], p[2]), d);

sort by values	Sort[d]			

functions

	mathematica	sympy	sage	maxima

define function	Add[a_, b_] := a + b
	
	(* alternate syntax: *)
	Add = Function[{a, b}, a + b]
				add(a, b) := a + b;
				
				define(add(a, b), a + b);
				
				/* block body: */
				add(a, b) := block(print("adding", a, "and", b), a + b);
				
				/* square bracket syntax: */
				I[row, col] := if row = col then 1 else 0;
				I[10, 10];

invoke function	Add[3, 7] Add @@ {3, 7} (* syntax for unary functions: *) 2 // Log			add(3, 7);
boolean function attributes *list, set, clear*	Attributes[add] SetAttributes[add, {Orderless, Flat, Listable}] ClearAtttibutes[add, Listable]			

undefine function	Clear[Add]			remfunction(add);
redefine function	Add[a_, b_] := b + a			add(a, b) := b + a;
missing function behavior	*The expression is left unevaluated. The head is the function name as a symbol, and the parts are the arguments.*			*The expression is left unevaluated.*
missing argument behavior	*The expression is left unevaluated. The head is the function name as a symbol, and the parts are the arguments.*			*Too few arguments error.*
extra argument behavior	*The expression is left unevaluated. The head is the function name as a symbol, and the parts are the arguments.*			*Too many arguments error.*
default argument	Options[myLog] = {base -> 10} myLog[x_, OptionsPattern[]] := N[Log[x]/Log[OptionValue[base]]] (* call using default: *) myLog[100] (* override default: *) myLog[100, base -> E]			
return value	*last expression evaluated, or argument of* Return[]			*last expression evaluated* *Inside a* block()*, the last expression evaluated or the argument of* return()

anonymous function	Function[{a, b}, a + b]
	
	(#1 + #2) &
				f: lambda([x, y], x + y);
				
				f(3, 7);

variable number of arguments	(* one or more arguments: *)
	add[a__] := Plus[a]
	
	(* zero or more arguments: *)
	add[a___] := Plus[a]
				add([a]) := sum(a[i], i, 1, length(a));

pass array elements as separate arguments	Apply[f, {a, b, c}]
	
	f @@ {x, y, z}
		a = [x, y, z]
		f(*a)
				add(a, b) := a + b;
				apply(add, [3, 7]);

execution control				
	mathematica	**sympy**	**sage**	**maxima**

if	If[x > 0,
	  Print["positive"],
	  If[x < 0,
	    Print["negative"],
	    Print["zero"]]]
		use Python execution control	*use* Python execution control	if x > 0
				  then print("positive")
				  else if x < 0
				    then print("negative")
				    else print("zero");

while	i = 0 While[i < 10, Print[i]; i++]			for i: 0 step 1 while i < 10 do print(i);
for	For[i = 0, i < 10, i++, Print[i]]			for i: 1 step 1 thru 10 do print(i);
break	Break[]			
continue	Continue[]			
exceptions				
	mathematica	**sympy**	**sage**	**maxima**
raise exception	Throw["failed"]	*use* Python exceptions	*use* Python exceptions	error("failed");
handle exception	Print[Catch[Throw["failed"]]]			errcatch(error("failed"));
streams				
	mathematica	**sympy**	**sage**	**maxima**
standard file handles	Streams["stdout"] Streams["stderr"] (* all open file handles: *) Streams[]			
write line to stdout	Print["hello"]			
open file for reading	f = OpenRead["/etc/hosts"]			
open file for writing	f = OpenWrite["/tmp/test"]			
open file for appending	f = OpenAppend["/tmp/test"]			
close file	Close[f]			

read file into string	s = ReadString[f]			
write string	WriteString[f, "lorem ipsum"]			

read file into array of strings	s = Import["/etc/hosts"]
	a = StringSplit[s, "\n"]			

file handle position *get, set*	f = StringToStream["foo bar baz"] StreamPosition[f] (* beginning of stream: *) SetStreamPosition[f, 0] (# end of stream: *) SetStreamPosition[f, Infinity]			

open temporary file	f = OpenWrite[]
	path = Part[f, 1]			

files

	mathematica	sympy	sage	maxima
file exists test, regular file test	FileExistsQ["/etc/hosts"] FileType["/etc/hosts"] == File			
file size	FileByteCount["/etc/hosts"]			
is file readable, writable, executable				
last modification time	FileDate["/etc/hosts"]			
copy file, remove file, rename file	CopyFile["/tmp/foo", "/tmp/bar"] DeleteFile["/tmp/foo"] RenameFile["/tmp/bar", "/tmp/foo"]			

directories

	mathematica	sympy	sage	maxima
working directory	dir = Directory[] SetDirectory["/tmp"]			
build pathname	FileNameJoin[{"/etc", "hosts"}]			
dirname and basename	DirectoryName["/etc/hosts"] FileBaseName["/etc/hosts"]			
absolute pathname	(* file must exist; symbolic links are resolved: *) AbsoluteFileName["foo"] AbsoluteFileName["/foo"] AbsoluteFileName["../foo"] AbsoluteFileName["./foo"] AbsoluteFileName["~/foo"]			
glob paths	Function[x, Print[x]] /@ FileNames["/tmp/*"]			
make directory	CreateDirectory["/tmp/foo.d"]			
recursive copy	CopyDirectory["/tmp/foo.d", "/tmp/baz.d"]			
remove empty directory	DeleteDirectory["/tmp/foo.d"]			
remove directory and contents	DeleteDirectory["/tmp/foo.d", DeleteContents -> True]			
directory test	DirectoryQ["/etc"]			

libraries and namespaces

	mathematica	sympy	sage	maxima
load library	Get["foo.m"]			load(grobner);

reflection

	mathematica	sympy	sage	maxima

get function documentation	?Tan
	Information[Tan]
		print(solve.__doc__)
		
		# in IJupyter:
		solve?
		help(solve)
			solve?	describe(solve);
				
				? solve;

function options	Options[Solve] Options[Plot]			

function source		import inspect
		
		inspect.getsourcelines(integrate)		

query data type	`Head[x]`		`type(x)`	`symbolp(x);`
				`numberp(7);`
				`stringp("seven");`
				`listp([1, 2, 3]);`

list variables in scope	`Names[$Context <> "*"]`			`/* user defined variables: */`
				`values;`
				
				`/* user defined functions: */`
				`functions;`

sheet two: symbolic expressions | calculus | equations and unknowns | optimization | vectors | matrices | combinatorics | number theory | polynomials | trigonometry | special functions | permutations | descriptive statistics | distributions | statistical tests

bar charts | scatter plots | line charts | surface charts | chart options

version used

The version of software used to check the examples in the reference sheet.

show version

How to determine the version of an installation.

implicit prologue

Code assumed to have been executed by the examples in the sheet.

Grammar and Invocation

interpreter

How to execute a script.

mathematica:

The full path to MathKernel on Mac OS X:

```
/Applications/Mathematica.app/Contents/MacOS/MathKernel
```

repl

How to launch a command line read-eval-print loop for the language.

block delimiters

How blocks are delimited.

statement separator

How statements are separated.

end-of-line comment

Character used to start a comment that goes to the end of the line.

multiple line comment

The syntax for a delimited comment which can span lines.

Variables and Expressions

assignment

How to perform assignment.

Mathematica, Sympy, and Pari/GP support the chaining of assignments. For example, in Mathematica one can assign the value 3 to `x` and `y` with:

```
x = y = 3
```

In Mathematica and Pari/GP, assignments are expressions. In Mathematica, the following code is legal and evaluates to 7:

```
(x = 3) + 4
```

In Mathematica, the `Set` function behaves identically to assignment and can be nested:

```
Set[a, Set[b, 3]]
```

delayed assignment

How to assign an expression to a variable name. The expression is re-evaluated each time the variable is used.

mathematica:

GNU make also supports assignment and delayed assignment, but `=` is used for delayed assignment and `:=` is used for immediate assignment. This is the opposite of how Mathematica uses the symbols.

The POSIX standard for make only has `=` for delayed assignment.

parallel assignment

How to assign values in parallel.

Parallel assignment can be used to swap the values held in two variables.

compound assignment

The compound assignment operators.

increment and decrement

Increment and decrement operators which can be used in expressions.

non-referential identifier

An identifier which does not refer to a value.

A non-referential identifier will usually print as a string containing its name.

Expressions containing non-referential identifiers will not be evaluated, though they may be simplified.

Non-referential identifiers represent "unknowns" or "parameters" when performing algebraic derivations.

identifier as value

How to get a value referring to an identifier.

The identifier may be the name of a variable containing a value. But the value referring to the identifier is distinct from the value in the variable.

One may manipulate a value referring to an identifier even if it is not the name of a variable.

global variable

How to declare a global variable.

local variable

How to declare a local variable.

pari/gp:

There is `my` for declaring a local variable with lexical scope and `local` for declaring a variable with dynamic scope.

`local` can be used to change the value of a global as seen by any functions which are called while the local scope is in effect.

null

The null literal.

null test

How to test if a value is null.

undefined variable access

What happens when an undefined variable is used in an expression.

remove variable binding

How to remove a variable. Subsequent references to the variable will be treated as if the variable were undefined.

conditional expression

A conditional expression.

Arithmetic and Logic

true and false

The boolean literals.

falsehoods

Values which evaluate to false in a conditional test.

sympy:

Note that the logical operators `Not`, `And` and `Or` do not treat empty collections or `None` as false. This is different from the Python logical operators `not`, `and`, and `or`.

pari/gp:

A vector or matrix evaluates to false if all components evaluate to false.

logical operators

The Boolean operators.

sympy:

In Python, `&`, `|`, and `&` are bit operators. SymPy has defined __and__, __or__, and __invert__ methods to make them Boolean operators for symbols, however.

relational operators

The relational operators.

sympy:

The full SymPy names for the relational operators are:

```
sympy.Equality             # ==
sympy.Unequality           # !=
sympy.GreaterThan          # >=
sympy.LessThan             # <=
sympy.StrictGreaterThan    # >
sympy.StrictLessThan       # <
```

The SymPy functions are attatched to the relational operators `==`, `!=`, for symbols … using the methods __eq__, __ne__, __ge__, __le__, __gt__, __lt__. The behavior they provide is similar to the default Python behavior, but when one of the arguments is a SymPy expression, a simplification will be attempted before the comparison is made.

arithmetic operators

The arithmetic operators.

integer division

How to compute the quotient of two integers.

integer division by zero

The result of dividing an integer by zero.

float division

How to perform float division, even if the arguments are integers.

float division by zero

The result of dividing a float by zero.

power

How to compute exponentiation.

Note that zero to a negative power is equivalent to division by zero, and negative numbers to a fractional power may have multiple complex solutions.

sqrt

The square root function.

For positive arguments the positive square root is returned.

sqrt -1

How the square root function handles negative arguments.

mathematica:

An uppercase `I` is used to enter the imaginary unit, but Mathematica displays it as a lowercase `i`.

transcendental functions

The standard transcendental functions such as one might find on a scientific calculator.

The functions are the exponential (not to be confused with exponentiation), natural logarithm, sine, cosine, tangent, arcsine, arccosine, arctangent, and the two argument arctangent.

transcendental constants

The transcendental constants *pi* and *e*.

The transcendental functions can used to computed to compute the transcendental constants:

```
pi = acos(-1)
pi = 4 * atan(1)
e = exp(1)
```

float truncation

Ways to convert a float to a nearby integer.

absolute value

How to get the absolute value and signum of a number.

integer overflow

What happens when the value of an integer expression cannot be stored in an integer.

The languages in this sheet all support arbitrary length integers so the situation does not happen.

float overflow

What happens when the value of a floating point expression cannot be stored in a float.

rational construction

How to construct a rational number.

rational decomposition

How to extract the numerator and denominator from a rational number.

decimal approximation

How to get a decimal approximation of an irrational number or repeating decimal rational.

complex construction

How to construct a complex number.

complex decomposition

How to extract the real and imaginary part from a complex number; how to extract the argument and modulus; how to get the complex conjugate.

random number

How to generate a random integer or a random float.

pari/gp:

When the argument of `random()` is an integer `n`, it generates an integer in the range $$\{0, ..., n - 1\}$$.

When the argument is a arbitrary precision float, it generates a value in the range `[0.0, 1.0]`. The precision of the argument determines the precision of the random number.

random seed

How to set or get the random seed.

mathematica:

The seed is not set to the same value at start up.

bit operators

binary, octal, and hex literals

Binary, octal, and hex integer literals.

mathematica:

The notation works for any base from 2 to 36.

radix

Convert a number to a representation using a given radix.

to array of digits

Convert a number to an array of digits representing the number.

Strings

string literal

The syntax for a string literal.

newline in literal

Are newlines permitted in string literals.

literal escapes

Escape sequences for putting unusual characters in string literals.

concatenate

How to concatenate strings.

translate case

How to convert a string to all lower case letters or all upper case letters.

trim

How to remove whitespace from the beginning or the end of string.

number to string

How to convert a number to a string.

string to number

How to parse a number from a string.

string join

How to join an array of strings into a single string, possibly separated by a delimiter.

split

How to split a string in to an array of strings. How to specify the delimiter.

substitute

How to substitute one or all occurrences of substring with another.

length

How to get the length of a string in characters.

index of substring

How to get the index of the first occurrence of a substring.

extract substring

How to get a substring from a string using character indices.

character literal

The syntax for a character literal.

character lookup

How to get a character from a string by index.

chr and ord

Convert a character code point to a character or a single character string.

Get the character code point for a character or single character string.

delete characters

Delete all occurrences of a set of characters from a string.

Arrays

section	mathematica	maple	maxima	sympy
arrays	`List`	`list`	`list`	`list`
multidimensional arrays	`List`	`Array`	`array`	*none*
vectors	`List`	`Vector`	`list`	`Matrix`
matrices	`List`	`Matrix`	`matrix`	`Matrix`

literal

The notation for an array literal.

size

The number of elements in the array.

lookup

How to access an array element by its index.

update

How to change the value stored at an array index.

out-of-bounds behavior

What happens when an attempt is made to access an element at an out-of-bounds index.

element index

How to get the index of an element in an array.

slice

How to extract a subset of the elements. The indices for the elements must be contiguous.

array of integers as index

manipulate back

manipulate front

head

tail

cons

concatenate

replicate

copy

How to copy an array. Updating the copy will not alter the original.

iterate

reverse

sort

dedupe

membership

How to test whether a value is an element of a list.

intersection

How to to find the intersection of two lists.

union

How to find the union of two lists.

relative complement, symmetric difference

How to find all elements in one list which are not in another; how to find all elements which are in one of two lists but not both.

map

filter

reduce

universal and existential tests

min and max element

shuffle and sample

How to shuffle an array. How to extract a random sample from an array without replacement.

flatten

zip

How to interleave two arrays.

cartesian product

Sets

Arithmetic Sequences

Dictionaries

record literal

record member access

Functions

definition

invocation

function value

Execution Control

if

How to write a branch statement.

mathematica:

The 3rd argument (the else clause) of an *If* expression is optional.

while

How to write a conditional loop.

mathematica:

Do can be used for a finite unconditional loop:

```
Do[Print[foo], {10}]
```

for

How to write a C-style for statement.

break/continue

How to break out of a loop. How to jump to the next iteration of a loop.

Exceptions

raise exception

How to raise an exception.

handle exception

How to handle an exception.

uncaught exception behavior

gap:

Calling `Error()` invokes the GAP debugger, which is similar to a Lisp debugger. In particular, all the commands available in the GAP REPL are still available. Variables can be inspected and modified while in the debugger but any changes will be lost when the debugger is quitted.

One uses `quit;` or `^D` to exit the debugger. These commands also cause the top-level GAP REPL exit if used while not in a debugger.

If `Error()` is invoked while in the GAP debugger, the debugger will be invoked recursively. One must use `quit;` for each level of debugger recursion to return to the top -level GAP REPL.

Use

```
brk> Where(4);
```

to print the top four functions on the stack when the error occurred. Use `DownEnv()` and `UpEnv()` to move down the stack—i.e. from callee to caller—and `UpEnv()` to move up the stack. The commands take the number of levels to move down or up:

```
brk> DownEnv(2);
brk> UpEnv(2);
```

When the debugger is invoked, it will print a message. It may give the user the option of providing a value with the `return` statement so that a computation can be continued:

```
brk> return 17;
```

finally block

How to write code that executes even if an exception is raised.

Streams

Files

Directories

Libraries and Namespaces

Reflection

function documentation

How to get the documentation for a function.

Mathematica

Mathematica Documentation Center
WolframAlpha
Mathics

Maple

http://www.maplesoft.com/support/help/

Maxima

http://maxima.sourceforge.net/docs/manual/maxima.html

Sage

http://doc.sagemath.org/html/en/index.html

SymPy

Welcome to SymPy's documentation!