Hyperpolyglot

Numerical Analysis & Statistics: MATLAB, R, NumPy, Julia

a side-by-side reference sheet

	matlab	r	numpy	julia
version used	*MATLAB 8.3* *Octave 3.8*	*3.1*	*Python 2.7* *NumPy 1.7* *SciPy 0.13* *Pandas 0.12* *Matplotlib 1.3*	*0.4*

show version	`$ matlab -nojvm -nodisplay -r 'exit'`
	
	`$ octave --version`
		`$ R --version`	`sys.version`
			`np.__version__`
			`sp.__version__`
			`mpl.__version__`
				`$ julia --version`

implicit prologue	*none*	`install.packages('ggplot2')`
		`library('ggplot2')`
			`import sys, os, re, math`
			`import numpy as np`
			`import scipy as sp`
			`import scipy.stats as stats`
			`import pandas as pd`
			`import matplotlib as mpl`
			`import matplotlib.pyplot as plt`	

		grammar and invocation		
	matlab	r	numpy	julia

interpreter	`$ cat >>foo.m`
	`1 + 1`
	`exit`
	
	`$ matlab -nojvm -nodisplay -r "run('foo.m')"`
	
	`$ octave foo.m`
		`$ cat >>foo.r`
		`1 + 1`
		
		`$ Rscript foo.r`
		
		`$ R -f foo.r`
			`$ cat >>foo.py`
			`print(1 + 1)`
			
			`$ python foo.py`
				`$ cat >>foo.jl`
				`println(1 + 1)`
				
				`$ julia foo.jl`

repl	`$ matlab -nojvm -nodisplay`
	
	`$ octave`
		`$ R`	`$ python`	`$ julia`

command line program	`$ matlab -nojvm -nodisplay -r 'disp(1 + 1); exit'`
	
	`$ octave --silent --eval '1 + 1'`
		`$ Rscript -e 'print("hi")'`	`python -c 'print("hi")'`	`$ julia -e 'println("hi")'`

block delimiters	`function end`
	`if elseif else end`
	`while end`
	`for end`
		`{ }`	*offside rule*	

statement separator	`;` *or newline* *Newlines not separators after three dots:* `...` *Output is suppressed when lines end with a semicolon.*	`;` *or sometimes newline* *Newlines not separators inside (), [], {}, '', "", or after binary operator.*	*newline or ;* *Newlines not separators inside (), [], {}, triple quote literals, or after backslash: \*	
end-of-line comment	`1 + 1 % addition`	`1 + 1 # addition`	`1 + 1 # addition`	`1 + 1 # addition`
		variables and expressions		
	matlab	r	numpy	julia

assignment	`i = 3`	`i = 3`
		`i <- 3`
			`i = 3`	`i = 3`

		3 -> i assign("i", 3)		
parallel assignment				
swap				
compound assignment *arithmetic, string, logical*	*none*	*none*	# do not return values: += -= *= /= //= %= **= += *= &= |= ^=	
null	% Only used in place of numeric values: NaN	NA NULL	None np.nan # None cannot be stored in a numpy array; # np.nan can if dtype is float64.	# Only used in place of float values: NaN

null test	isnan(v)
	
	% true for '', []:
	isempty(v)
		is.na(v)
		is.null(v)
			v == None
			v is None
			
			np.isnan(np.nan)
			# np.nan == np.nan is False
				isnan(v)

conditional expression	*none*	(if (x > 0) x else -x) ifelse(x > 0, x, -x)	x if x > 0 else -x	x > 0 ? x : -x

arithmetic and logic

	matlab	r	numpy	julia
boolean type			# Python: bool # NumPy: bool_	Bool
true and false	1 0 true false	TRUE FALSE T F	True False	true false
falsehoods	false 0 0.0 *matrices evaluate to false unless nonempty and all entries evaluate to true*	FALSE F 0 0.0 *matrices evaluate to value of first entry; string in boolean context causes error*	False None 0 0.0 '' [] {}	false
logical operators	~true | (true & false) % short-circuit operators: && ||	!TRUE | (TRUE & FALSE) *short-circuit operators:* && || *& and | can operate on and return vectors, but && and || return scalars*	and or not	&& || !
relational operators	== ~= > < >= <=	== != > < >= <=	== != > < >= <=	== != > < >= <=
integer type			# Python: int # NumPy: int8 int16 int32 int64	Int8 Int16 Int32 Int64 Int128 Int *is either 32 or 64 bits, depending on* WORD_SIZE
unsigned type			uint8 uint16 uint32 uint64	UInt8 UInt16 UInt32 UInt64 UInt128 UInt *is either 32 or 64 bits, depending on* WORD_SIZE
float type			# Python: float # NumPy: float16 float32 float64	Float16 Float32 Float64

arithmetic operators
add, sub, mult, div, quot, rem
	+ - * / *none* mod(*n*, *divisor*)	+ - * / %/% %%	+ - * / // %	+ - * / div(*n*, *divisor*) rem(*n*, *divisor*)
				
				# always non-negative:
				mod(*n*, *divisor*)

integer division	fix(13 / 5)	13 %/% 5 as.integer(13 / 5)	13 // 5	div(13, 5)
integer division by zero	Inf NaN *or* -Inf	*result of converting Inf or NaN to an integer with as.integer:* NA	*raises* ZeroDivisionError	*raises* DivideError

float division	13 / 5	13 / 5	float(13) / 5	13 / 5 5 \ 13
float division by zero *dividend is positive, zero, negative*	*these values are literals:* Inf NaN -Inf	*these values are literals:* Inf NaN -Inf	*raises* ZeroDivisionError	*these values are literals:* Inf NaN -Inf
power	2 ^ 16	2 ^ 16 2 ** 16	2 ** 16	2 ^ 16
sqrt	sqrt(2)	sqrt(2)	math.sqrt(2)	sqrt(2)

sqrt -1	% returns 0 + 1i:
	sqrt(-1)
		# returns NaN:
		sqrt(-1)
		
		# returns 0+1i:
		sqrt(-1+0i)
			# raises ValueError:
			math.sqrt(-1)
			
			# returns 1.41421j:
			import cmath
			cmath.sqrt(-1)
				# raises DomainError:
				sqrt(-1)
				
				# returns 0.0 + 1.0im:
				sqrt(-1 + 0im)

transcendental functions	exp log sin cos tan asin acos atan atan2	exp log sin cos tan asin acos atan atan2	math.exp math.log math.sin math.cos math.tan math.asin math.acos math.atan math.atan2	exp log sin cos tan asin acos atan atan2
transcendental constants	pi e	pi exp(1)	math.pi math.e	pi e

float truncation
round towards zero, to nearest integer, down, up
	fix(x)
	round(x)
	floor(x)
	ceil(x)
		as.integer(x)
		round(x)
		floor(x)
		ceiling(x)
			int(x)
			int(round(x))
			math.floor(x)
			math.ceil(x)
				Int(trunc(x))
				Int(round(x))
				Int(floor(x))
				Int(ceil(x))
				
				# trunc() and other functions return floats.
				# Int() raises InexactError if float argument has
				# nonzero fractional portion.

absolute value
and signum
	abs sign	abs sign	abs(-3.7)
			math.copysign(1, -3.7)
				abs(-3.7)
				sign(-3.7)

integer overflow	*becomes float; largest representable integer in the variable* intmax	*becomes float; largest representable integer in the variable* .Machine$integer.max	*becomes arbitrary length integer of type* long	
float overflow	Inf	Inf	*raises* OverflowError	
float limits	eps realmax realmin	.Machine$double.eps .Machine$double.xmax .Machine$double.xmin	np.finfo(np.float64).eps np.finfo(np.float64).max np.finfo(np.float64).min	
rational construction				22 // 7

rational decomposition				num(22 // 7)
				den(22 // 7)

complex types			complex64 complex128	Complex32 Complex64 Complex128
complex construction	1 + 3i	1 + 3i	1 + 3j	1 + 3im complex(1, 3)

complex decomposition	real imag
	abs arg
	conj
		Re Im
		abs Arg
		Conj
			import cmath
			
			z.real
			z.imag
			cmath.polar(z)[1]
				real(1 + 3im)
				imag(1 + 3im)
				abs(1 + 3im)
				angle(1 + 3im)
				conj(1 + 3im)

random number
uniform integer, uniform float
	floor(100 * rand)
	rand
		floor(100 * runif(1))
		runif(1)
			np.random.randint(0, 100)
			np.random.rand()
				rand(1:100)
				rand()

random seed
set, get, and restore
	rand('state', 17)
	sd = rand('state')
	rand('state', sd)
		set.seed(17)
		sd = .Random.seed
		none
			np.random.seed(17)
			sd = np.random.get_state()
			np.random.set_state(sd)	

bit operators	bitshift(100, 3)
	bitshift(100, -3)
	bitand(1, 2)
	bitor(1, 2)
	bitxor(1, 2)
	% MATLAB:
		none	100 << 3
			100 >> 3
			1 & 2
			1 | 2
			1 ^ 2
			~1	

	bitcmp(1, 'uint16')
	% Octave:
	bitcmp(1, 16)			

binary, octal, and hex literals				0b101010 0o52 0x2a

radix
convert integer to and from string with radix
				base(7, 42)
				parse(Int, "60", 7)

strings				
	matlab	**r**	**numpy**	**julia**
literal	'don''t say "no"' % Octave only: "don't say \"no\""	"don't say \"no\"" 'don\'t say "no"'	'don\'t say "no"' "don't say \"no\"" r"don't " r'say "no"'	"don't say \"no\""
newline in literal	*no*	*yes*	*no*	*yes*
literal escapes	% Octave double quote only: \\ \" \' \0 \a \b \f \n \r \t \v	\\ \" \' \a \b \f \n \r \t \v \\ooo	# single and double quoted: \newline \\ \' \" \a \b \f \n \r \t \v \\ooo \x*hh*	\\ \" \' \a \b \f \n \t \r \v \ooo \x*hh* \u*hhhh* \U*hhhhhhhh*

variable interpolation				count = 3
				item = "ball"
				println("$count $(item)s")

expression interpolation				"1 + 1 = $(1 + 1)"
concatenate	strcat('one ', 'two ', 'three')	paste("one", "two", "three", sep=" ")	'one ' + 'two ' + 'three' *literals, but not variables, can be concatenated with juxtaposition:* 'one ' "two " 'three'	"one " * "two " * "three" string("one ", "two ", "three")

replicate	hbar = repmat('-', 1, 80)	hbar = paste(rep('-', 80), collapse='')	hbar = '-' * 80	hbar = "-" ^ 80
				
				hbar = repeat("-", 80)

index of substring	% returns array of one-indexed
	% locations
	strfind('hello', 'el')
		counts from one, returns -1 if not found
		regexpr("el", "hello")
			# Counts from zero; raises ValueError if not found:
			'hello'.index('el')
				# returns UnitRange:
				search("hello", "el")

extract substring	s = 'hello'
	% syntax error: 'hello'(1:4)
	s(1:4)
		substr("hello", 1, 4)	'hello'[0:4]	"hello"[1:4]

split	% returns cell array: strsplit('foo,bar,baz', ',')	strsplit('foo,bar,baz', ',')	'foo,bar,baz'.split(',')	split("foo,bar,baz", ",")

join	% takes cell array as arg:
	strjoin({'foo', 'bar', 'baz'}, ',')
		paste("foo", "bar", "baz", sep=",")
		paste(c('foo', 'bar', 'baz'),
		  collapse=',')
			','.join(['foo', 'bar', 'baz'])	join(["foo", "bar", "baz"], ",")

trim
both sides, left, right
	strtrim(' foo ')
	none
	deblank('foo ')
		gsub("(^[\n\t]+|[\n\t]+$)",
		  "",
		  " foo ")
		sub("^[\n\t]+", "", " foo")
		sub("[\n\t]+$", "", "foo ")
			' foo '.strip()
			' foo'.lstrip()
			'foo '.rstrip()	

pad
on right, on left, centered
	s = repmat(' ', 1, 10)
	s(1:5) = 'lorem'
	strjust(s, 'left')
	strjust(s, 'right')
	strjust(s, 'center')
		sprintf("%-10s", "lorem")
		sprintf("%10s", "lorem")
		none
			'lorem'.ljust(10)
			'lorem'.rjust(10)
			'lorem'.center(10)	

number to string	strcat('value: ', num2str(8))	paste("value: ", toString("8"))	'value: ' + str(8)	
string to number	7 + str2num('12') 73.9 + str2num('.037')	7 + as.integer("12") 73.9 + as.double(".037")	7 + int('12') 73.9 + float('.037')	7 + parse(Int, "12") 73.9 + parse(Float64, ".037")

translate case	lower('FOO')
	upper('foo')
		tolower("FOO")
		toupper("foo")
			'foo'.upper()
			'FOO'.lower()
				uppercase("foo")
				lowercase("FOO")

sprintf	sprintf('%s: %.3f %d', 'foo', 2.2, 7)	sprintf("%s: %.3f %d", "foo", 2.2, 7)	'%s: %.3f %d' % ('foo', 2.2, 7)	@sprintf("%s: %.2f %d", "foo", 2.2, 7)

length	length('hello')	nchar("hello")	len('hello')	length("hello")
				
				# index of first byte of last char:
				endof("hello")

character access	s = 'hello'
	% syntax error: 'hello'(1)
	s(1)
		substr("hello", 1, 1)	'hello'[0]	"hello"[1]
				
				# index must be byte-index of the first byte of a
				# character. Raises BoundsErrror if no such byte,
				# and UnicodeError if byte not first in char.

chr and ord	char(65)
	double('A')
		intToUtf8(65)
		utf8ToInt("A")
			chr(65)
			ord('A')
				Char(65)
				Int('A')

		regular expressions		
	matlab	**r**	**numpy**	**julia**
character class abbreviations	. \d \D \s \S \w \W % also C-string style backslash escapes: \a \b \f \n \r \t \v	# escape backslash in strings by doubling: . \d \D \s \S \w \W	. \d \D \s \S \w \W	. \d \D \h \H \s \S \v \V \w \W
anchors	^ $ \< \>	# escape backslash in strings by doubling: ^ $ \< \> \b \B	^ $ \A \b \B \Z	^ $ \A \b \B \z \Z

match test	regexp('hello', '^[a-z]+$')
	regexp('hello', '^\S+$')
		regexpr("^[a-z]+$", "hello") > 0
		regexpr('^\\S+$', "hello") > 0
			re.search(r'^[a-z]+$', 'hello')
			re.search(r'^\S+$', 'hello')
				ismatch(r"^[a-z]+$", "hello")

case insensitive match test	regexpi('Lorem Ipsum', 'lorem')	regexpr('(?i)lorem', "Lorem Ipsum") > 0	re.search(r'lorem', 'Lorem Ipsum', re.I)	ismatch(r"lorem"i, "Lorem Ipsum")
modifiers	*none*	(?i) (?m) (?s) (?x)	re.I re.M re.S re.X	i m s x

substitution
first match, all matches
	s = 'do re mi mi mi'
	regexprep(s, 'ma', 'once')
	regexprep(s, 'mi', 'ma')
		sub('mi', 'ma', 'do re mi mi mi')
		gsub('mi', 'ma', 'do re mi mi mi')
			rx = re.compile(r'mi')
			s = rx.sub('ma', 'do re mi mi mi', 1)
			s2 = rx.sub('ma', 'do re mi mi mi')
				replace("do re mi mi mi", r"mi", s"ma", 1)
				replace("do re mi mi mi", r"mi", s"ma")

backreference in match and substitution	regexp('do do', '(\w+) \1')
	regexprep('do re', '(\w+) (\w+)', '$2 $1')
		regexpr('(\\w+) \\1', 'do do') > 0
		sub('(\\w+) (\\w+)', '\\2 \\1', 'do re')
			none
			
			rx = re.compile(r'(\w+) (\w+)')
			rx.sub(r'\2 \1', 'do re')
				ismatch(r"(\w+) \1", "do do")

group capture			rx = '(\d{4})-(\d{2})-(\d{2})'
			m = re.search(rx, '2010-06-03')
			yr, mo, dy = m.groups()
				rx = r"(\d{4})-(\d{2})-(\d{2})"
				m = match(rx, "2010-06-03")
				yr, mn, dy = m.captures

		dates and time		
	matlab	**r**	**numpy**	**julia**

current date/time	t = now	t = as.POSIXlt(Sys.time())	import datetime
			
			t = datetime.datetime.now()
				t = now()

date/time type	*floating point number representing days since year 0 in the Gregorian calendar*	POSIXlt	datettime	DateTime
date/time difference type	*floating point number representing days*	*a* difftime *object which behaves like a floating point number representing seconds*	timedelta, *which can be converted to float value in seconds via* total_seconds() *method*	Base.Dates.Millisecond

get date parts	dv = datevec(t)
	dv(1)
	dv(2)
	dv(3)
	% syntax error: datevec(t)(1)
		t$year + 1900
		t$mon + 1
		t$mday
			t.year
			t.month
			t.day
				Dates.year(t)
				Dates.month(t)
				Dates.day(t)

get time parts	dv = datevec(t)
	dv(4)
	dv(5)
	dv(6)
		t$hour
		t$min
		t$sec
			t.hour
			t.minute
			t.second
				Dates.hour(t)
				Dates.minute(t)
				Dates.second(t)

build date/time from parts	t = datenum([2011 9 20 23 1 2])	t = as.POSIXlt(Sys.time())
		t$year = 2011 - 1900
		t$mon = 9 - 1
		t$mday = 20
		t$hour = 23
		t$min = 1
		t$sec = 2
			import datetime
			
			t = datetime.datetime(2011, 9, 20, 23, 1, 2)
				t = DateTime(2011, 9, 20, 23, 1, 2)

convert to string	datestr(t)	print(t)	str(t)	"$t"

parse datetime	s = '2011-09-20 23:01:02'
	fmt = 'yyyy-mm-dd HH:MM:SS'
	t = datenum(s, fmt)
		t = strptime('2011-09-20 23:01:02',
		  '%Y-%m-%d %H:%M:%S')
			import datetime
			
			s = '2011-05-03 10:00:00'
			fmt = '%Y-%m-%d %H:%M:%S'
			t = datetime.datetime.strptime(s, fmt)
				fmt = "yyyy-mm-dd HH:MM:SS"
				t = DateTime("2011-05-03 10:00:00", fmt)
				
				# fmt string can be compiled:
				df = Dates.DateFormat(fmt)
				t2 = DateTime("2011-05-03 10:00:00", df)

format datetime	datestr(t, 'yyyy-mm-dd HH:MM:SS')	format(t, format='%Y-%m-%d %H:%M:%S')	t.strftime('%Y-%m-%d %H:%M:%S')	Dates.format(t, "yyyy-mm-dd HH:MM:SS")

sleep	pause(0.5)	Sys.sleep(0.5)	import time
			
			time.sleep(0.5)
				sleep(0.5)

tuples				
	matlab	**r**	**numpy**	**julia**
type	cell	list	tuple	Tuple{T[, ...]}
literal	tup = {1.7, 'hello', [1 2 3]}	tup = list(1.7, "hello", c(1, 2, 3))	tup = (1.7, "hello", [1,2,3])	tup = (1.7, "foo", [1, 2, 3])
lookup element	% indices start at one: tup{1}	# indices start at one: tup[[1]]	# indices start at zero: tup[0]	# indices start at one: tup[1]
update element	tup{1} = 2.7	tup[[1]] = 2.7	*tuples are immutable*	*tuples are immutable*
length	length(tup)	length(tup)	len(tup)	length(tup)
arrays				
	matlab	**r**	**numpy**	**julia**

element type	*always numeric*	# "numeric":
		class(c(1, 2, 3))
		
		# arrays can also have "boolean" or "string" elements
			# values can have different types:
			[type(x) for x in a]
				a = [1, 2, 3]
				
				# Array{Int64, 2}:
				typeof(a)
				# Int64:
				typeof(a[1])

literal	a = [1, 2, 3, 4]
	
	% commas are optional:
	a = [1 2 3 4]
		# use c() constructor:
		a = c(1, 2, 3, 4)
			a = [1, 2, 3, 4]	a = [1, 2, 3, 4]

size	length(a)	length(a)	len(a)	length(a)
empty test	length(a) == 0 % An array used in a conditional test is % false unless nonempty and all entries evaluate % as true.	length(a) == 0	not a	isempty(a)
lookup	% Indices start at one: a(1)	# Indices start at one: a[1]	# Indices start at zero: a[0]	# Indices start at one: a[1]
update	a(1) = -1	a[1] = -1	a[0] = -1	a[1] = -1

out-of-bounds behavior	a = []
	
	% error:
	a(1)
	
	% increases array size to 10;
	% zero-fills slots 1 through 9:
	a(10) = 10
		a = c()
		# evaluates as NA:
		a[10]
		# increases array size to 10:
		a[10] = "lorem"
			a = []
			# raises IndexError:
			a[10]
			# raises IndexError:
			a[10] = 'lorem'
				a = []
				
				# raises BoundsError:
				a[10]
				# raises BoundsError:
				a[10] = "lorem"

index of element	a = [7 8 9 10 8]
	
	% returns [2 5]:
	find(a == 8)
		a = c('x', 'y', 'z', 'w', 'y')
		
		# c(2, 5):
		which(a == 'y')
			a = ['x', 'y', 'z', 'w', 'y']
			
			a.index('y') # 1
			a.rindex('y') # 4
				a = ["x", "y", "z", "w", "y"]
				
				# 2:
				findfirst(a, "y")

	`% returns 2:`
	`find(a == 8, 1, 'first')`
				`# 5:`
				`julia> findlast(a, "y")`

slice
by endpoints
	`a = ['a' 'b' 'c' 'd' 'e']`
	
	`% ['c' 'd']:`
	`a(3:4)`
		`a = c("a", "b", "c", "d", "e")`
		
		`# c("c", "d"):`
		`a[seq(3, 4)]`
			`a = ['a', 'b', 'c', 'd', 'e']`
			
			`# ['c', 'd']:`
			`a[2:4]`
				`a = ["a", "b", "c", "d", "e"]`
				
				`# ["c", "d"]:`
				`a[3:4]`

slice to end	`a = ['a' 'b' 'c' 'd' 'e']`
	
	`% ['c' 'd' 'e']:`
	`a(3:end)`
		`a = c("a", "b", "c", "d", "e")`
		
		`# both return c("c", "d", "e"):`
		`tail(a, n=length(a) - 2)`
		`a[-1:-2]`
			`a = ['a', 'b', 'c', 'd', 'e']`
			
			`# ['c', 'd', 'e']:`
			`a[2:]`
				`a = ["a", "b", "c", "d", "e"]`
				
				`# ["c", "d", "e"]:`
				`a[3:end]`

integer array as index	`[7 8 9]([1 3 3])`	`c(7, 8, 9)[c(1, 3, 3)]`	`np.array([7, 8, 9])[[0, 2, 2]]`	`# [7, 9, 9]:`
				`[7, 8, 9][[1, 3, 3]]`

logical array as index	`[7 8 9]([true false true])`	`c(7, 8, 9)[c(T, F, T)]`	`np.array([7, 8, 9])[[True, False, True]]`	`# [7, 9]:`
				`[7, 8, 9][[true, false, true]]`

concatenate	`a = [1 2 3]`
	`a2 = [a [4 5 6]]`
	`a = [a [4 5 6]]`
	`% or:`
	`a = horzcat(a, a2)`
		`a = c(1, 2, 3)`
		`a2 = append(a, c(4, 5, 6))`
		`a = append(a, c(4, 5, 6))`
			`a = [1, 2, 3]`
			`a2 = a + [4, 5, 6]`
			`a.extend([4, 5, 6])`
				`a = [1, 2, 3]`
				`a2 = vcat(a, [4, 5, 6])`
				`a = vcat(a, [4, 5, 6])`

replicate	`a = repmat(NA, 1, 10)`	`a = rep(NA, 10)`
		
		`# 30 a's, 50 b's, and 90 c's:`
		`rep(c("a", "b", "c"), c(30, 50, 90))`
			`a = [None] * 10`
			`a = [None for i in range(0, 10)]`
				`fill(NaN, 10)`

copy
address copy, shallow copy, deep copy
	There is no address copy. Because arrays cannot be nested, there is no distinction between shallow copy and deep copy. Assignment and passing an array to a function can be regarded as performing a shallow or deep copy, though MATLAB does not allocate memory for a 2nd array until one of the arrays is modified.	*Arrays in R behave like arrays in MATLAB.*	`import copy`
			
			`a = [1, 2, [3, 4]]`
			
			`a2 = a`
			`a3 = list(a)`
			`a4 = copy.deepcopy(a)`
				`a = Any[1, 2, [3, 4]]`
				
				`a2 = a`
				`a3 = copy(a)`
				`a4 = deepcopy(a)`

iteration	`a = [9 7 3]`
	`for i = 1:numel(a)`
	`  x = a(i)`
	`  disp(x)`
	`end`
		`for (x in c(9, 7, 3)) {`
		`  print(x)`
		`}`
			`for i in [9, 7, 3]:`
			`  print(i)`
				`for i = [9, 7, 3]`
				`  println(i)`
				`end`

indexed iteration		`for (i in seq_along(a)) {`
		`  cat(sprintf("%s at index %d\n", i, a[i]))`
		`}`
			`a = ['do', 're', 'mi', 'fa']`
			`for i, s in enumerate(a):`
			`  print('%s at index %d' % (s, i))`
				`a = ["do", "re", "mi", "fa"]`
				`for (i, s) in enumerate(a)`
				`  println(i, " ", s)`
				`end`

reverse	`a = [1 2 3]`
	`a2 = fliplr(a)`
	`a = fliplr(a)`
		`a = c(1, 2, 3)`
		`a2 = rev(a)`
		`a = rev(a)`
			`a = [1, 2, 3]`
			`a2 = a[::-1]`
			`a.reverse()`
				`a = [1, 2, 3]`
				`a2 = reverse(a)`
				`reverse!(a)`

sort	`a = [3 1 4 2]`
	`a = sort(a)`
		`a = c('b', 'A', 'a', 'B')`
		`a2 = sort(a)`
		`a = sort(a)`
			`a = ['b', 'A', 'a', 'B']`
			`sorted(a)`
			`a.sort()`
			`a.sort(key=str.lower)`
				`a = [3, 1, 4, 2]`
				`a2 = sort(a)`
				`sort!(a)`

dedupe	`a = [1 2 2 3]`
	`a2 = unique(a)`
		`a = c(1, 2, 2, 3)`
		`a2 = unique(a)`
			`a = [1, 2, 2, 3]`
			`a2 = list(set(a))`
				`a = unique([1, 2, 2, 3])`

membership	`ismember(7, a)`	`7 %in% a`
		`is.element(7, a)`
			`7 in a`	`7 in a`
				`7 ∈ a`
				`a ∋ 7`

intersection	`intersect([1 2], [2 3 4])`	`intersect(c(1, 2), c(2, 3, 4))`	`{1, 2} & {2, 3, 4}`	`intersection([1, 2], [2, 3, 4])`
				`∩([1, 2], [2, 3, 4])`

union	`union([1 2], [2 3 4])`	`union(c(1, 2), c(2, 3, 4))`	`{1, 2} | {2, 3, 4}`	`union([1, 2], [2, 3, 4])`
				`∪([1, 2], [2, 3, 4])`

relative complement, symmetric difference	`setdiff([1 2 3], [2])`
	
	`a1 = [1 2]`
	`a2 = [2 3 4]`
	`union(setdiff(a1, a2), setdiff(a2, a1))`
		`setdiff(c(1, 2, 3), c(2))`
		
		`union(setdiff(c(1, 2), c(2, 3, 4)),`
		`  setdiff(c(2, 3, 4), c(1, 2)))`
			`{1, 2, 3} - {2}`
			
			`{1, 2} ^ {2, 3, 4}`
				`setdiff([1, 2, 3], [2])`
				`symdiff([1, 2], [2, 3, 4])`

map	`arrayfun( @(x) x*x, [1 2 3])`	`sapply(c(1,2,3), function (x) { x * x})`	`map(lambda x: x * x, [1, 2, 3])`
			`# or use list comprehension:`
			`[x * x for x in [1, 2, 3]]`	

filter	a = [1 2 3]
	a(a > 2)
		a = c(1, 2, 3)
		a[a > 2]
		
		Filter(function(x) { x > 2}, a)
			filter(lambda x: x > 1, [1, 2, 3])
			# or use list comprehension:
			[x for x in [1, 2, 3] if x > 1]	

reduce		Reduce(function(x, y) { x + y }, c(1, 2, 3), 0)	reduce(lambda x, y: x + y, [1 ,2, 3], 0)	reduce(+, [1, 2, 3])
				foldl(-, 0, [1, 2, 3])
				foldr(-, 0, [1, 2, 3])

universal and existential tests	all(mod([1 2 3 4], 2) == 0)
	any(mod([1 2 3 4]) == 0)
		all(c(1, 2, 3, 4) %% 2 == 0)
		any(c(1, 2, 3, 4) %% 2 == 0)
			all(i % 2 == 0 for i in [1, 2, 3, 4])
			any(i % 2 == 0 for i in [1, 2, 3, 4])
				all([x % 2 == 0 for x in [1, 2, 3, 4]])
				any([x % 2 == 0 for x in [1, 2, 3, 4]])

shuffle and sample		a = c(1, 1, 2, 3, 9, 28)
		sample(a, 3)
		a[sample.int(length(a))]
			from random import shuffle, sample
			
			a = [1, 2, 3, 4]
			shuffle(a)
			sample(a, 2)	

zip	*none; MATLAB arrays can't be nested*	# R arrays can't be nested.
		# One approximation of zip is a 2d array:
		a = rbind(c(1, 2, 3), c('a', 'b', 'c'))
		
		# To prevent data type coercion, use a data frame:
		df = data.frame(numbers=c(1, 2, 3),
		  letters=c('a', 'b', 'c'))
			# array of 3 pairs:
			a = zip([1, 2, 3], ['a', 'b', 'c'])	

arithmetic sequences

	matlab	r	numpy	julia

unit difference	1:100	# type integer:
		1:100
		seq(1, 100)
		
		# type double:
		seq(1, 100, 1)
			range(1, 101)	1:100

difference of 10	0:10:100	# type double:
		seq(0, 100, 10)
			range(0, 101, 10)	0:10:100

difference of 0.1	0:0.1:10	seq(0, 10, 0.1)	[0.1 * x for x in range(0, 101)] # 3rd arg is length of sequence, not step size: sp.linspace(0, 10, 100)	0:0.1:10

computed difference	% 100 evenly spaced values:
	linspace(3.7, 19.4, 100)
	
	% 100 is default num. of elements:
	linspace(3.7, 19.4)
			numpy.linspace(3.7, 19.4, 100)	

iterate			# range replaces xrange in Python 3:
			n = 0;
			for i in xrange(1, 1000001):
			  n += i
				n = 0
				for i in 1:1000000
				  n += i
				end

to array			a = range(1, 11)
			# Python 3:
			a = list(range(1, 11))
				a = Array(1:10)

two dimensional arrays

	matlab	r	numpy	julia

element type	*always numeric*	A = array(c(1, 2, 3, 4), dim=c(2, 2))
		
		# "array":
		class(A)
		
		# "boolean", "numeric", or "string":
		class(c(A))
			np.array([[1, 2], [3, 4]]).dtype
			
			# possible values: np.bool, np.int64,
			# np.float64, np.complex128, ...
				A = [1 2; 3 4]
				
				eltype(A)

literal	[1, 2; 3, 4] % commas optional; newlines can replace semicolons:: [1 2 3 4]	*none*	*none*	[1 2; 3 4] # A 1-d array created with commas is a # n×1 array. If commas are used in a literal, # then semicolons and spaces as delimiters. [1 2 3 4]

construct from sequence	reshape([1 2 3 4], 2, 2)	array(c(1, 2, 3, 4), dim=c(2, 2))	A = np.array([1, 2, 3, 4]).reshape(2, 2)
			
			# convert to nested Python lists:
			A.tolist()
				reshape([1, 2, 3, 4], 2, 2)

construct from rows	row1 = [1 2 3]
	row2 = [4 5 6]
	
	A = [row1; row2]
		rbind(c(1, 2, 3), c(4, 5, 6))	row1 = np.array([1, 2, 3])
			row2 = np.array([4, 5, 6])
			
			np.vstack((row1, row2))
			
			np.array([[1, 2], [3, 4]])
				vcat([1 2 3], [4 5 6])
				
				row1 = [1 2 3]
				row2 = [4 5 6]
				[row1; row2]

construct from columns	col1 = [1; 4]
	col2 = [2; 5]
	col3 = [3; 6]
	
	% commas are optional:
	A = [col1, col2, col3]
		cbind(c(1, 4), c(2, 5), c(3, 6))	cols = (
			  np.array([1, 4]),
			  np.array([2, 5]),
			  np.array([3, 6])
			)
			np.vstack(cols).transpose()
				hcat([1, 4], [2, 5], [3, 6])
				
				col1 = [1, 4]
				col2 = [2, 5]
				col3 = [3, 6]
				[col1 col2 col3]

construct from subarrays	A = [1 3; 2 4]
	
	A4_by_2 = [A; A]
	A2_by_4 = [A A]
		A = matrix(c(1, 2, 3, 4), nrow=2)
		A4_by_2 = rbind(A, A)
		A2_by_4 = cbind(A, A)
			A = np.array([[1, 2], [3, 4]])
			A2_by_4 = np.hstack([A, A])
			A4_by_2 = np.vstack([A, A])
				A = [1 2; 3 4]
				A4_by_2 = [A; A]
				A2_by_4 = [A A]

cast element type				

size
number of elements, number of dimensions, dimension lengths
	numel(A)
	ndims(A)
	size(A)
	
	% length of 1st dimension (i.e. # of rows):
	size(A, 1)
	
	% length of longest dimension:
	length(A)
		length(A)
		length(dim(A))
		dim(A)
			A.size
			A.ndim
			A.shape
			
			# number of rows:
			len(A)
				length(A)
				ndims(A)
				size(A)

lookup	% indices start at one:
	[1 2; 3 4](1, 1)
		# indices start at one:
		A = array(c(1, 2, 3, 4), dim=c(2, 2)
		
		A[1, 1]
			# indices start at zero:
			A = np.array([[1, 2], [3, 4]])
			
			A[0][0] *or*
			A[0, 0]
				# indices start at one:
				A[1, 1]

1d lookup	A = [2 4; 6 8]
	% returns 8:
	A(4)
	
	% convert to column vector of length 4:
	A2 = A(:)
		A = array(c(2, 4, 6, 8), dim=c(2, 2))
		
		# returns 8:
		A[4]
			A = np.array([[2, 4], [6, 8]])
			
			# returns np.array([6, 8]):
			A[1]
			
			# returns 8:
			A.flat[3]
				A = [2 4; 6 8]
				
				# returns 8:
				A[4]

lookup row or column	A = [1 2 3; 4 5 6; 7 8 9]
	
	% 2nd row:
	A(2, :)
	
	% 2nd column:
	A(:, 2)
		A = t(array(1:9, dim=c(3, 3)))
		
		# 2nd row:
		A[2,]
		
		# 2nd column:
		A[, 2]
			A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
			
			# 2nd row:
			A[1, :]
			
			# 2nd column:
			A[:, 1]
				A = [1 2 3; 4 5 6; 7 8 9]
				
				# 2nd row:
				A[2, :]
				
				# 2nd column:
				A[:, 2]

update	A = [2 4; 6 8] A(1, 1) = 3	A = array(c(2, 4, 6, 8), dim=c(2, 2)) A[1, 1] = 3	A = np.array([[2, 4], [6, 8]]) A[0, 0] = 3	A = [2 4; 6 8] A[1, 1] = 3

update row or column	A = [1 2; 3 4]
	
	% [2 1; 3 4]:
	A(1, :) = [2 1]
	
	% [3 1; 2 4]:
	A(:, 1) = [3 2]
		A = t(array(1:4, dim=c(2, 2)))
		
		A[1,] = c(2, 1)
		A[, 1] = c(3, 2)
			A = np.array([[1, 2], [3, 4]])
			
			A[0, :] = [2, 1]
			# or
			A[0] = [2, 1]
			
			A[:, 0] = [3, 2]
				A = [1 2; 3 4]
				
				A[1, :] = [2 1]
				A[:, 1] = [3; 2]

update subarray	A = ones(3, 3)
	A(1:2, 1:2) = 2 * ones(2, 2)
	% or just:
	A(1:2, 1:2) = 2
		A = array(1, dim=c(3, 3))
		A[1:2, 1:2] = array(2, dim=c(2, 2))
		# or just:
		A[1:2, 1:2] = 2
			A = np.ones([3, 3])
			A[0:2, 0:2] = 2 * np.ones([2, 2])
				A = ones(3, 3)
				A[1:2, 1:2] = 2 * ones(2, 2)

out-of-bounds behavior	A = [2 4; 6 8]
	
	% error:
	x = A(3, 1)
		Lookups and updates both cause subscript out of bounds error.	*Lookups and updates both raise an* IndexError *exception.*	BoundsError

	% becomes 3x2 array with zero at (3, 2): A(3, 1) = 9			

slice subarray	A = reshape(1:16, 4, 4)'
	
	% top left 2x2 subarray:
	A(1:2, 1:2)
	
	% bottom right 2x2 subarray:
	A(end-1:end, end-1:end)
	
	% 2x2 array containing corners:
	A([1 4], [1 4])
	A([1 end], [1 end])
		A = t(array(1:16, dim=c(4, 4)))
		
		# top left 2x2 subarray:
		A[1:2, 1:2]
		
		# bottom right 2x2 subarray:
		A[-1:-2, -1:-2]
		
		# 2x2 array containing corners:
		A[c(1, 4), c(1, 4)]
			A = np.array(range(1, 17)).reshape(4, 4)
			
			# top left 2x2 subarray:
			A[0:2, 0:2]
			
			# bottom right 2x2 subarray:
			A[2:, 2:]
				A = reshape(1:16, 4, 4)
				
				A[1:2, 1:2]
				
				A[3:4, 3:4]

transpose	A = [1 2; 3 4]
	
	transpose(A)
		A = array(c(1, 2, 3, 4), dim=c(2, 2))
		t(A)
			A = np.array([[1, 2], [3, 4]])
			A.transpose()
			A.T
				A = [1 2; 3 4]
				
				transpose(A)

flip	% [2 1; 4 3]:
	fliplr([1 2; 3 4])
	
	% [3 4; 1 2]:
	flipud([1 2; 3 4])
		# install.packages('pracma'):
		require(pracma)
		
		A = t(array(1:4, dim=c(2, 2)))
		
		fliplr(A)
		flipud(A)
			A = np.array([[1, 2], [3, 4]])
			
			np.fliplr(A)
			np.flipud(A)
				# [2 1; 4 3]:
				flipdim([1 2; 3 4], 2)
				
				# [3 4; 1 2]:
				flipdim([1 2; 3 4], 1)

circular shift
 	
along columns, along rows
	A = [1 2; 3, 4]
	
	% [3 4; 1 2]:
	circshift(A, 1)
	
	% [2 1; 4 3]:
	circshift(A, 1, 2)
	
	% The 2nd argument can be any integer; negative values shift
	% in the opposite direction.
		# install.packages('pracma'):
		require(pracma)
		
		A = t(array(1:4, dim=c(2, 2)))
		
		circshift(A, c(1, 0))
		circshift(A, c(0, 1))
			A = np.array([[1, 2], [3, 4]])
			
			np.roll(A, 1, axis=0)
			np.roll(A, 1, axis=1)
				circshift([1 2; 3 4], [1, 0])
				circshift([1 2; 3 4], [0, 1])

rotate
clockwise, counter-clockwise
	A = [1 2; 3 4]
	
	% [3 1; 4 2]:
	rot90(A, -1)
	
	% [2 4; 1 3]:
	rot90(A)
	
	% set 2nd arg to 2 for 180 degree rotation
		# install.packages('pracma'):
		require(pracma)
		
		A = t(array(1:4, dim=c(2, 2)))
		
		rot90(A)
		rot90(A, -1)
		rot90(A, 2)
			A = np.array([[1, 2], [3, 4]])
			
			np.rot90(A)
			np.rot90(A, -1)
			np.rot90(A, 2)
				A = [1 2; 3 4]
				
				rotr90(A)
				rotl90(A)
				rotr90(A, 2)

reduce
rows, columns
	M = [1 2; 3 4]
	
	% sum each row:
	cellfun(@sum, num2cell(M, 2))
	
	% sum each column:
	cellfun(@sum, num2cell(M, 1))
	
	% sum(M, 2) and sum(M, 1) also sum rows and columns
		M = matrix(c(1, 2, 3, 4), nrow=2)
		
		# sum each row:
		apply(M, 1, sum)
		
		# sum each column:
		apply(M, 2, sum)
			M = np.array([[1, 2], [3, 4]])
			
			np.add.reduce(A, 1)
			
			np.add.reduce(A, 0)
			
			# np.add is a built-in universal function. All universal functions have a reduce method.
			
			# np.sum(A, 1,) and np.sum(A, 0) also sum rows and columns
				A = [1 2; 3 4]
				
				[3; 7]:
				reducedim(+, A, [2], 0)
				
				[4 6]:
				reducedim(+, A, [1], 0)

		three dimensional arrays		
	matlab	**r**	**numpy**	**julia**
construct from sequence	reshape([1 2 3 4 5 6 7 8], 2, 2, 2)	array(seq(1, 8), dim=c(2, 2, 2))	np.array(range(1, 9)).reshape(2, 2, 2)	reshape(1:8, 2, 2, 2)
construct from nested sequences	*none*	*none*	np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])	
construct 3d array from 2d arrays	A = [1, 2; 3, 4] A(:,:,2) = [5, 6; 7, 8]			A = Array{Float64}(2, 2, 2) A[:, :, 1] = [1 2; 3 4] A[:, :, 2] = [5 6; 7 8]

permute axes	A = reshape([1 2 3 4 5 6 7 8], 2, 2, 2)
	
	% swap 2nd and 3rd axes:
	permute(A, [1 3 2])
		A = array(1:8, dim=c(2, 2, 2))
		
		# swap 2nd and 3rd axes:
		aperm(A, perm=c(1, 3, 2))
			A = np.array(range(1, 9)).reshape(2, 2, 2)
			
			# swap 2nd and 3rd axes:
			A.transpose((0, 2, 1))
				A = reshape(1:8, 2, 2, 2)
				
				# swap 2nd and 3rd axes:
				reshape(A, [1, 3, 2])

flip	A = reshape([1 2 3 4 5 6 7 8], 2, 2, 2)	*none*	*none*	A = reshape(1:8, 2, 2, 2)

	flipdim(A, 3)			flipdim(A, 3)

circular shift	A = reshape([1 2 3 4 5 6 7 8], 2, 2, 2)
	
	% 3rd arg specifies axis:
	circshift(A, 1, 3)
		none	A = np.array(range(1, 9)).reshape(2, 2, 2)
			
			np.roll(A, 1, axis=2)
				A = reshape(1:8, 2, 2, 2)
				
				circshift(A, [0, 0, 1])

dictionaries				
	matlab	r	numpy	julia

literal	% no literal; use constructor:
	d = struct('n', 10, 'avg', 3.7, 'sd', 0.4)
	
	% or build from two cell arrays:
	d = cell2struct({10 3.7 0.4}, {'n' 'avg' 'sd'}, 2)
		# keys are 'n', 'avg', and 'sd':
		d = list(n=10, avg=3.7, sd=0.4)
		
		# keys are 1, 2, and 3:
		d2 = list('do', 're', 'mi')
			d = {'n': 10, 'avg': 3.7, 'sd': 0.4}	d = Dict("n"=>10.0, "avg"=>3.7, "sd"=>0.4)

size	length(fieldnames(d))	length(d)	len(d)	length(d)
lookup	d.n getfield(d, 'n')	d[['n']] # if 'n' is a key: d$n	d['n']	
update	d.var = d.sd**2	d$var = d$sd**2	d['var'] = d['sd']**2	
missing key behavior	*error*	NULL	*raises* KeyError	
is key present	isfield(d, 'var')	is.null(d$var)	'var' in d	haskey(d, "var")
delete	d = rmfield(d, 'sd')	d$sd = NULL	del(d['sd'])	

iterate	for i = 1:numel(fieldnames(d))
	  k = fieldnames(d){i}
	  v = d.(k)
	  code
	end
		for (k in names(d)) {
		  v = d[[k]]
		  code
		}
			for k, v in d.iteritems():
			  code	

keys and values as arrays	% these return cell arrays:
	fieldnames(d)
	struct2cell(d)
		names(d)
		unlist(d, use.names=F)
			d.keys()
			d.values()	

merge	*none*	d1 = list(a=1, b=2)
		d2 = list(b=3, c=4)
		# values of first dictionary take precedence:
		d3 = c(d1, d2)
			d1 = {'a':1, 'b':2}
			d2 = {'b':3, 'c':4}
			d1.update(d2)	

functions				
	matlab	r	numpy	julia

define function	function add(x, y)
	  x + y
	end
		add = function(x, y) {x + y}		function add(x,y)
				  x + y
				end
				
				# optional syntax when body is an expression:
				add(x, y) = x + y

invoke function	add(3, 7)	add(3, 7)		add(3, 7)

nested function	function ret1 = add3(x, y, z)
	  function ret2 = add2(x, y)
	    ret2 = x + y;
	  end
	
	  ret1 = add2(x, y) + z;
	end
		add3 = function(x, y, z) {
		  add2 = function(x, y) { x + y }
		  add2(x, y) + z
		}
				function add3(x, y, z)
				  function add2(x2, y2)
				    x2 + y2
				  end
				  add2(x, y) + z
				end

missing argument behavior	*raises error if code with the parameter that is missing an argument is executed*	*raises error*		*raises* MethodError
extra argument behavior	*ignored*	*raises error*		*raises* MethodError

default argument	function mylog(x, base=10)
	  log(x) / log(base)
	end
		mylog = function(x,base=10) {
		  log(x) / log(base)
		}		

variadic function	function s = add(varargin)
	  if nargin == 0
	    s = 0
	  else
	    r = add(varargin{2:nargin})
	    s = varargin{1} + r
	  end
	end
		add = function (...) {
		  a = list(...)
		  if (length(a) == 0)
		    return(0)
		  s = 0
		  for(i in 1:length(a)) {
		    s = s + a[[i]]
		  }
		  return(s)
		}		

return value	function ret = add(x, y) ret = x + y; end % If a return variable is declared, the % value assigned to it is returned. Otherwise % the value of the last statement will be % used if it does not end with a semicolon.	return *argument or last expression evaluated.* NULL *if* return *called without an argument.*		return *argument or last expression evaluated.* Void *if* return *called without an argument.*

multiple return values	function [x, y] = first_two(a)
	  x = a(1);
	  y = a(2);
	end
	
	% sets first to 7; second to 8:
	[first, second] = first_two([7 8 9])
				function first_two(a)
				  a[1], a[2]
				end
				
				x, y = first_two([1, 2, 3])

anonymous function literal	% body must be an expression:
	@(x, y) x + y
		function(x, y) {x + y}		add = (x, y) -> x + y
				
				add = function(x, y)
				  x + y
				end

invoke anonymous function				add(1, 2)

closure		make_counter = function() {
		  i = 0
		  function() {
		    i <<- i + 1
		    i
		  }
		}		

function as value	@add	add		add
overload operator				
call operator like function		\`+\`(3, 7)		+(3, 7)

execution control

	matlab	r	numpy	julia

if	if (x > 0)
	  disp('positive')
	elseif (x < 0)
	  disp('negative')
	else
	  disp('zero')
	end
		if (x > 0) {
		  print('positive')
		} else if (x < 0) {
		  print('negative')
		} else {
		  print('zero')
		}
			if x > 0:
			  print('positive')
			elif x < 0:
			  print('negative')
			else:
			  print('zero')
				if x > 0
				  println("positive")
				elseif x < 0
				  println("negative")
				else
				  println("zero")
				end

while	i = 0
	while (i < 10)
	  i = i + 1
	  disp(i)
	end
		while (i < 10) {
		  i = i + 1
		  print(i)
		}
			while i < 10:
			  i += 1
			  print(i)
				i = 0
				while i < 10
				  i += 1
				  println(i)
				end

for	for i = 1:10
	  disp(i)
	end
		for (i in 1:10) {
		  print(i)
		}
			for i in range(1,11):
			  print(i)
				for i = 1:10
				  println(i)
				end

break/continue	break continue	break next	break continue	break continue
raise exception	error('%s', 'failed')	stop('failed')	raise Exception('failed')	throw("failed")

handle exception	try
	  error('failed')
	catch err
	  disp(err)
	end
		tryCatch(
		  stop('failed'),
		  error=function(e) print(message(e)))
			try:
			  raise Exception('failed')
			except Exception as e:
			  print(e)	

		file handles		
	matlab	**r**	**numpy**	**julia**
standard file handles	0 1 2 % Octave has predefined variables % containing the above descriptors: stdin stdout stderr	stdin() stdout() stderr()	sys.stdin sys.stdout sys.stderr	STDIN STDOUT STDERR
read line from stdin	line = input('', 's')	line = readLines(n=1)	line = sys.stdin.readline()	line = readline()

write line to stdout	fprintf(1, 'hello\n')	cat("hello\n")
		
		writeLines("hello")
			print('hello')	println("hello")

write formatted string to stdout	fprintf(1, '%.2f\n', pi)	cat(sprintf("%.2f\n", pi))	import math
			
			print('%.2f' % math.pi)	

open file for reading	f = fopen('/etc/hosts')
	if (f == -1)
	  error('failed to open file')
	end
		f = file("/etc/hosts", "r")	f = open('/etc/hosts')	f = open("/etc/hosts")

open file for writing	if ((f = fopen('/tmp/test', 'w') == -1)
	  error('failed to open file')
	endif
		f = file("/tmp/test", "w")	f = open('/tmp/test', 'w')	f = open("/etc/hosts", "w")

open file for appending	if ((f = fopen('/tmp/err.log', 'a') == -1)
	  error('failed to open file')
	endif
		f = file("/tmp/err.log", "a")	f = open('/tmp/err.log', 'a')	f = open("/tmp/err.log", "a")

close file	fclose(f)	close(f)	f.close()	close(f)
i/o errors	fopen *returns -1;* fclose *throws an error*		*raise* IOError *exception*	
read line	line = fgets(f)	line = readLines(f, n=1)	line = f.readline()	line = readline(f)

iterate over file by line	while(!feof(f))
	  line = fgets(f)
	  puts(line)
	endwhile
			for line in f:
			  print(line)	

read file into array of strings		lines = readLines(f)	lines = f.readlines()	lines = readlines(f)
write string	fputs(f, 'lorem ipsum')	cat("lorem ipsum", file=f)	f.write('lorem ipsum')	write(f, "lorem ipsum")
write line	fputs(f, 'lorem ipsum\n')	writeLines("lorem ipsum", con=f)	f.write('lorem ipsum\n')	
flush file handle	fflush(f)	flush(f)	f.flush()	

file handle position
get, set
	ftell(f)
	
	% 3rd arg can be SEEK_CUR or SEEK_END
	fseek(f, 0, SEEK_SET)
		seek(f)
		
		# sets seek point to 12 bytes after start;
		# origin can also be "current" or "end"
		seek(f, where=0, origin="start")
			f.tell()
			
			f.seek(0)	

redirect stdout to file		sink("foo.txt")		

write variables to file	A = [1 2; 3 4]
	B = [4 3; 2 1]
	
	save('data.mdata', 'A', 'B')
		A = matrix(c(1, 3, 2, 4), nrow=2)
		B = matrix(c(4, 2, 3, 1), nrow=2)
		
		save(A, B, file='data.rdata')
			A = np.matrix([[1, 2], [3, 4]])
			B = np.matrix([[4, 3], [2, 1]])
			
			# Data must be of type np.array;
			# file will have .npz suffix:
			np.savez('data', A=A, B=B)	

read variables from file	% puts A and B in scope:
	load('data.mdata')
		# puts A and B in scope:
		load('data.rdata')
			data = np.load('data.npz')
			A = data['A']	

	% puts just A in scope: load('data.mdata', 'A')		B = data['B']	
write all variables in scope to file	save('data.txt')	save.image('data.txt')		
directories				
	matlab	**r**	**numpy**	**julia**

working directory
get, set
	pwd
	
	cd('/tmp')
		getwd()
		
		setwd("/tmp")
			os.path.abspath('.')
			
			os.chdir('/tmp')	

build pathname	fullfile('/etc', 'hosts')	file.path("/etc", "hosts")	os.path.join('/etc', 'hosts')	

dirname and basename	[dir, base] = fileparts('/etc/hosts')	dirname("/etc/hosts")
		basename("/etc/hosts")
			os.path.dirname('/etc/hosts')
			os.path.basename('/etc/hosts')	

absolute pathname		normalizePath("..")	os.path.abspath('..')	

iterate over directory by file	% lists /etc:
	ls('/etc')
	
	% lists working directory:
	ls()
		# list.files() defaults to working directory
		for (path in list.files('/etc')) {
		  print(path)
		}
			for filename in os.listdir('/etc'):
			  print(filename)	

glob paths	glob('/etc/*')	Sys.glob('/etc/*')	import glob
			
			glob.glob('/etc/*')	

processes and environment				
	matlab	**r**	**numpy**	**julia**

command line arguments	% does not include interpreter name:
	argv()
		# first arg is name of interpreter:
		commandArgs()
		
		# arguments after --args only:
		commandArgs(TRUE)
			sys.argv	ARGS

environment variable
get, set
	getenv('HOME')
	
	setenv('PATH', '/bin')
		Sys.getenv("HOME")
		
		Sys.setenv(PATH="/bin")
			os.getenv('HOME')
			
			os.environ['PATH'] = '/bin'
				ENV["HOME"]
				
				ENV["PATH"] = "/bin"

exit	exit(0)	quit(save="no", status=0)	sys.exit(0)	exit(0)

external command	if (shell_cmd('ls -l /tmp'))
	  error('ls failed')
	endif
		if (system("ls -l /tmp")) {
		  stop("ls failed")
		}
			if os.system('ls -l /tmp'):
			  raise Exception('ls failed')	

command substitution				s = readall(‘ls`)
libraries and namespaces				
	matlab	**r**	**numpy**	**julia**

load library	*When a function is invoked, MATLAB searches the library path for a file with the same name and a .m suffix. Other functions defined in the file are not visible outside the file.*	# quoting the name of the package is optional:
		require("foo")
		# or:
		library("foo")
		
		# if the package does not exist, require returns false, and library raises an error.
			import foo	include("foo.jl")

list loaded libraries	*none*	search()	dir()	

library search path	path()
	addath('~/foo')
	rmpath('~/foo')
		.libPaths()	sys.path	

source file	run('foo.m')	source("foo.r")	*none*	
install package	% Octave: how to install package % downloaded from Octave-Forge: pkg install foo-1.0.0.tar.gz	install.packages("ggplot2")	$ pip install scipy	

load package library	% Octave:
	pkg load foo
		require("foo")
		# or:
		library("foo")
			import foo	

list installed packages	`% Octave:`
	`pkg list`
		`library()`
		`installed.packages()`
			`$ pip freeze`	

reflection				
	matlab	**r**	**numpy**	**julia**

data type	`class(x)`	`class(x)`
		`# often more informative:`
		`str(x)`
			`type(x)`	`typeof(x)`

attributes	`% if x holds an object:`
	`x`
		`attributes(x)`	`[m for m in dir(x)`
			`  if not callable(getattr(o,m))]`
				`fieldnames(x)`

methods	`% if x holds an object:`
	`methods(x)`
		none; objects are implemented by functions which dispatch based on type of first arg	`[m for m in dir(x)`
			`  if callable(getattr(o,m))]`
				`methods(x)`

variables in scope	`who()`
	
	`% with size and type:`
	`whos()`
		`objects()`
		`ls()`
		
		`# with type and description:`
		`ls.str()`
			`dir()`	`whos()`

undefine variable	`clear('x')`	`rm(v)`	`del(x)`	*none*
undefine all variables	`clear -a`	`rm(list=objects())`		*none*
eval	`eval('1 + 1')`	`eval(parse(text='1 + 1'))`	`eval('1 + 1')`	`eval(parse("1 + 1"))`

function documentation	`help tan`	`help(tan)`
		`?tan`
			`math.tan.__doc__`	`?tan`

list library functions	*none*	`ls("package:moments")`	`dir(stats)`	`whos(Base)`
search documentation	`docsearch tan`	`??tan`	`$ pydoc -k tan`	`apropos("tan")`
debugging				
	matlab	**r**	**numpy**	**julia**

benchmark code	`tic`
	`n = 0`
	`for i = 1:1000*1000`
	`  n = n + 1;`
	`end`
	`toc`
			`import timeit`
			
			`timeit.timeit('i += 1',`
			`  'i = 0',`
			`  number=1000000)`	

tables | import and export | relational algebra | aggregation

vectors | matrices | sparse matrices | optimization | polynomials | descriptive statistics | distributions | linear regression | statistical tests | time series | fast fourier transform | clustering | images | sound

univariate charts | bivariate charts | multivariate charts

General

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 which examples in the sheet assume to have already been executed.

r:

The `ggplot2` library must be installed and loaded to use the plotting functions `qplot` and `ggplot`.

Grammar and Invocation

interpreter

How to invoke the interpreter on a script.

repl

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

r:

R installations come with a GUI REPL.

The shell `zsh` has a built-in command `r` which re-runs the last command. Shell built-ins take precedence over external commands, but one can invoke the R REPL with:

```
$ command r
```

command line program

How to pass the code to be executed to the interpreter as a command line argument.

environment variables

How to get and set an environment variable.

block delimiters

Punctuation or keywords which define blocks.

matlab:

The list of keywords which define blocks is not exhaustive. Blocks are also defined by

- *switch*, *case*, *otherwise*, *endswitch*
- *unwind_protect*, *unwind_protect_cleanup*, *end_unwind_protect*
- *try*, *catch*, *end_try_catch*

statement separator

How statements are separated.

matlab:

Semicolons are used at the end of lines to suppress output. Output echoes the assignment performed by a statement; if the statement is not an assignment the value of the statement is assigned to the special variable `ans`.

In Octave, but not MATLAB, newlines are not separators when preceded by a backslash.

end-of-line comment

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

octave:

Octave, but not MATLAB, also supports shell-style comments which start with `#`.

Variables and Expressions

assignment

r:

Traditionally <- was used in R for assignment. Using an = for assignment was introduced in version 1.4.0 sometime before 2002. -> can also be used for assignment:

```
3 -> x
```

compound assignment

The compound assignment operators.

octave:

Octave, but not MATLAB, has compound assignment operators for arithmetic and bit operations:

```
+= -= *= /=  **=  ^=
&= |=
```

Octave, but not MATLAB, also has the C-stye increment and decrement operators `++` and `--`, which can be used in prefix and postfix position.

increment and decrement operator

The operator for incrementing the value in a variable; the operator for decrementing the value in a variable.

null

matlab:

`NaN` can be used for missing numerical values. Using a comparison operator on it always returns false, including `NaN == NaN`. Using a logical operator on `NaN` raises an error.

octave:

Octave, but not MATLAB, provides `NA` which is a synonym of `NaN`.

r:

Relational operators return `NA` when one of the arguments is `NA`. In particular `NA == NA` is `NA`. When acting on values that might be `NA`, the logical operators observe the rules of ternary logic, treating `NA` is the unknown value.

null test

How to test if a value is null.

octave:

Octave, but not MATLAB, has `isna` and `isnull`, which are synonyms of `isnan` and `isempty`.

conditional expression

A conditional expression.

Arithmetic and Logic

true and false

The boolean literals.

matlab:

true and *false* are functions which return matrices of ones and zeros of type *logical*. If no arguments are specified they return single entry matrices. If one argument is provided, a square matrix is returned. If two arguments are provided, they are the row and column dimensions.

falsehoods

Values which evaluate to false in a conditional test.

matlab:

When used in a conditional, matrices evaluate to false unless they are nonempty and all their entries evaluate to true. Because strings are matrices of characters, an empty string ('' or "") will evaluate to false. Most other strings will evaluate to true, but it is possible to create a nonempty string which evaluates to false by inserting a null character; e.g. "false\000".

r:

When used in a conditional, a vector evaluates to the boolean value of its first entry. Using a vector with more than one entry in a conditional results in a warning message. Using an empty vector in a conditional, *c()* or *NULL*, raises an error.

logical operators

The boolean operators.

octave:

Octave, but not MATLAB, also uses the exclamation point '!' for negation.

relational operators

The relational operators.

octave:

Octave, but not MATLAB, also uses `!=` for an inequality test.

arithmetic operators

The arithmetic operators: addition, subtraction, multiplication, division, quotient, remainder.

matlab:

mod is a function and not an infix operator. *mod* returns a positive value if the first argument is positive, whereas *rem* returns a negative value.

integer division

How to compute the quotient of two integers.

integer division by zero

What happens when an integer is divided by zero.

float division

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

float division by zero

What happens when a float is divided by zero.

power

octave:

Octave, but not MATLAB, supports `**` as a synonym of `^`.

sqrt

The square root function.

sqrt(-1)

The result of taking the square root of a negative number.

transcendental functions

The standard transcendental functions.

transcendental constants

Constants for *pi* and *e*.

float truncation

Ways of converting a float to a nearby integer.

absolute value

The absolute value and signum of a number.

integer overflow

What happens when an expression evaluates to an integer which is too big to be represented.

float overflow

What happens when an expression evaluates to a float which is too big to be represented.

float limits

The machine epsilon; the largest representable float and the smallest (i.e. closest to negative infinity) representable float.

complex construction

Literals for complex numbers.

complex decomposition

How to decompose a complex number into its real and imaginary parts; how to decompose a complex number into its absolute value and argument; how to get the complex conjugate.

random number

How to generate a random integer from a uniform distribution; how to generate a random float from a uniform distribution.

random seed

How to set, get, and restore the seed used by the random number generator.

matlab:

At startup the random number generator is seeded using operating system entropy.

r:

At startup the random number generator is seeded using the current time.

numpy:

On Unix the random number generator is seeded at startup from /dev/random.

bit operators

The bit operators left shift, right shift, and, or , xor, and negation.

matlab/octave:

`bitshift` takes a second argument which is positive for left shift and negative for right shift.

`bitcmp` takes a second argument which is the size in bits of the integer being operated on. Octave is not compatible with MATLAB in how the integer size is indicated.

r:

There is a library on CRAN called `bitops` which provides bit operators.

Strings

literal

The syntax for a string literal.

newline in literal

Can a newline be included in a string literal? Equivalently, can a string literal span more than one line of source code?

octave:

Double quote strings are Octave specific.

A newline can be inserted into a double quote string using the backslash escape `\n`.

A double quote string can be continued on the next line by ending the line with a backslash. No newline is inserted into the string.

literal escapes

Escape sequences for including special characters in string literals.

matlab:

C-style backslash escapes are not recognized by string literals, but they are recognized by the IO system; the string 'foo\n' contains 5 characters, but emits 4 characters when written to standard output.

concatenate

How to concatenate strings.

replicate

How to create a string which consists of a character of substring repeated a fixed number of times.

index of substring

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

extract substring

How to get the substring at a given index.

octave:

Octave supports indexing string literals directly: `'hello'(1:4)`.

split

How to split a string into an array of substrings. In the original string the substrings must be separated by a character, string, or regex pattern which will not appear in the array of substrings.

The split operation can be used to extract the fields from a field delimited record of data.

matlab:

Cell arrays, which are essentially tuples, are used to store variable-length strings.

A two dimensional array of characters can be used to store strings of the same length, one per row. Regular arrays cannot otherwise be used to store strings.

join

How to join an array of substrings into single string. The substrings can be separated by a specified character or string.

Joining is the inverse of splitting.

trim

How to remove whitespace from the beginning and the end of a string.

Trimming is often performed on user provided input.

pad

How to pad the edge of a string with spaces so that it is a prescribed length.

number to string

How to convert a number to a string.

string to number

How to convert a string to number.

translate case

How to put a string into all caps. How to put a string into all lower case letters.

sprintf

How to create a string using a printf style format.

length

How to get the number of characters in a string.

character access

How to get the character in a string at a given index.

octave:

Octave supports indexing string literals directly: `'hello'(1)`.

chr and ord

How to convert an ASCII code to a character; how to convert a character to its ASCII code.

Regular Expressions

character class abbreviations

The supported character class abbreviations.

A character class is a set of one or more characters. In regular expressions, an arbitrary character class can be specified by listing the characters inside square brackets. If the first character is a circumflex `^`, the character class is all characters not in the list. A hyphen `-` can be used to list a range of characters.

matlab:

The C-style backslash escapes, which can be regarded as character classes which match a single character, are a feature of the regular expression engine and not string literals like in other languages.

anchors

The supported anchors.

The `\<` and `\>` anchors match the start and end of a word respectively.

match test

How to test whether a string matches a regular expression.

case insensitive match test

How to perform a case insensitive match test.

substitution

How to replace all substring which match a pattern with a specified string; how to replace the first substring which matches a pattern with a specified string.

backreference in match and substitution

How to use backreferences in a regex; how to use backreferences in the replacement string of substitution.

Date and Time

current date/time

How to get the current date and time.

r:

`Sys.time()` returns a value of type `POSIXct`.

date/time type

The data type used to hold a combined date and time value.

matlab:

The Gregorian calendar was introduced in 1582. The Proleptic Gregorian Calendar is sometimes used for earlier dates, but in the Proleptic Gregorian Calendar the year 1 CE is preceded by the year 1 BCE. The MATLAB epoch thus starts at the beginning of the year 1 BCE, but uses a zero to refer to this year.

date/time difference type

The data type used to hold the difference between two date/time types.

get date parts

How to get the year, the month as an integer from 1 through 12, and the day of the month from a date/time value.

octave:

In Octave, but not MATLAB, one can use index notation on the return value of a function:

```
t = now
datevec(t)(1)
```

get time parts

How to get the hour as an integer from 0 through 23, the minute, and the second from a date/time value.

build date/time from parts

How to build a date/time value from the year, month, day, hour, minute, and second as integers.

convert to string

How to convert a date value to a string using the default format for the locale.

parse datetime

How to parse a date/time value from a string in the manner of strptime from the C standard library.

format datetime

How to write a date/time value to a string in the manner of strftime from the C standard library.

Tuples

type

The name of the data type which implements tuples.

literal

How to create a tuple, which we define as a fixed length, inhomogeneous list.

lookup element

How to access an element of a tuple.

update element

How to change one of a tuple's elements.

length

How to get the number of elements in a tuple.

Arrays

This section covers one-dimensional arrays which map integers to values.

Multidimensional arrays are a generalization which map tuples of integers to values.

Vectors and matrices are one-dimensional and two-dimensional arrays respectively containing numeric values. They support additional operations including the dot product, matrix multiplication, and norms.

Here are the data types covered in each section:

section	matlab	r	numpy	julia
arrays	matrix (ndims = 1)	vector	list	
multidimensional arrays	matrix	array	np.array	
vectors	matrix (ndims = 1)	vector	np.array (ndim = 1)	
matrices	matrix (ndims = 2)	matrix	np.matrix	

element type

How to get the type of the elements of an array.

permitted element types

Permitted data types for array elements.

matlab:

Arrays in Octave can only contain numeric elements.

Array literals can have a nested structure, but Octave will flatten them. The following literals create the same array:

```
[ 1 2 3 [ 4 5 6] ]
[ 1 2 3 4 5 6 ]
```

Logical values can be put into an array because *true* and *false* are synonyms for 1 and 0. Thus the following literals create the same arrays:

```
[ true false false ]
[ 1 0 0 ]
```

If a string is encountered in an array literal, the string is treated as an array of ASCII values and it is concatenated with other ASCII values to produce as string. The following literals all create the same string:

```
[ 'foo', 98, 97, 114]
[ 'foo', 'bar' ]
'foobar'
```

If the other numeric values in an array literal that includes a string are not integer values that fit into a ASCII byte, then they are converted to byte sized values.

r:

Array literals can have a nested structure, but R will flatten them. The following literals produce the same array of 6 elements:

```
c(1,2,3,c(4,5,6))
c(1,2,3,4,5,6)
```

If an array literal contains a mixture of booleans and numbers, then the boolean literals will be converted to 1 (for TRUE and T) and 0 (for FALSE and F).

If an array literal contains strings and either booleans or numbers, then the booleans and numbers will be converted to their string representations. For the booleans the string representations are "TRUE'" and "FALSE".

literal

The syntax, if any, for an array literal.

matlab:

The array literal

```
[1,'foo',3]
```

will create an array with 5 elements of class *char*.

r:

The array literal

```
c(1,'foo',3)
```

will create an array of 3 elements of class *character*, which is the R string type.

size

How to get the number of values in an array.

empty test

lookup

update

out-of-bounds behavior

index of element

slice

slice to end

concatenate

replicate

copy

How to make an address copy, a shallow copy, and a deep copy of an array.

After an address copy is made, modifications to the copy also modify the original array.

After a shallow copy is made, the addition, removal, or replacement of elements in the copy does not modify of the original array. However, if elements in the copy are modified, those elements are also modified in the original array.

A deep copy is a recursive copy. The original array is copied and a deep copy is performed on all elements of the array. No change to the contents of the copy will modify the contents of the original array.

Arithmetic Sequences

An arithmetic sequence is a sequence of numeric values in which consecutive terms have a constant difference.

unit difference

An arithmetic sequence with a difference of 1.

difference of 10

An arithmetic sequence with a difference of 10.

difference of 0.1

An arithmetic sequence with a difference of 0.1.

computed difference

An arithmetic sequence where the difference is computed using the start and end values and the number of elements.

iterate

How to iterate over an arithmetic sequence.

to array

How to convert an arithmetic sequence to an array.

Multidimensional Arrays

Multidimensional arrays are a generalization of arrays which map tuples of integers to values. All tuples in the domain of a multidimensional array have the same length; this length is the dimension of the array.

The multidimensional arrays described in this sheet are homogeneous, meaning that the values are all of the same type. This restriction allows the implementation to store the values of the multidimensional array in a contiguous region of memory without the use of references or points.

Multidimensional arrays should be contrasted with nested arrays. When arrays are nested, the innermost nested arrays contain the values and the other arrays contain references to arrays. The syntax for looking up a value is usually different:

```
# nested:
a[1][2]

# multidimensional:
a[1, 2]
```

element type

How to get the type of the values stored in a multidimensional array.

r:

literal—1d

literal—2d

construct from sequence—2d

construct from sequence—3d

construct from nested sequences—2d

construct from nested sequences—3d

construct from rows—2d

construct from columns—2d

construct from subarrays—2d

construct 3d array from 2d arrays

size

lookup—1d

lookup—2d

1d lookup on 2d array

update—2d

out-of-bounds behavior

slice

slice to end

slice subarray

transpose

permute axes

flip—2d

flip—3d

circular shift—2d

rotate—2d

apply function element-wise

apply function to linear subarrays

Dictionaries

literal

The syntax for a dictionary literal.

size

How to get the number of keys in a dictionary.

lookup

How to use a key to lookup a value in a dictionary.

update

How to add or key-value pair or change the value for an existing key.

missing key behavior

What happens when looking up a key that isn't in the dictionary.

delete

How to delete a key-value pair from a dictionary.

iterate

How to iterate over the key-value pairs.

keys and values as arrays

How to get an array containing the keys; how to get an array containing the values.

merge

How to merge two dictionaries.

Functions

define function

How to define a function.

invoke function

How to invoke a function.

nested function

missing argument behavior

What happens when a function is invoked with too few arguments.

extra argument behavior

What happens when a function is invoked with too many arguments.

default argument

How to assign a default argument to a parameter.

variadic function

How to define a function which accepts a variable number of arguments.

return value

How the return value of a function is determined.

multiple return values

How to return multiple values from a function.

anonymous function literal

The syntax for an anonymous function.

invoke anonymous function

closure

function as value

How to store a function in a variable.

Execution Control

if

How to write a branch statement.

while

How to write a conditional loop.

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.

raise exception

How to raise an exception.

handle exception

How to handle an exception.

File Handles

standard file handles

Standard input, standard output, and standard error.

read line from stdin

write line to stdout

How to write a line to stdout.

matlab:

The backslash escape sequence `\n` is stored as two characters in the string and interpreted as a newline by the IO system.

write formatted string to stdout

open file for reading

open file for writing

open file for appending

close file

i/o errors

read line

iterate over file by line

read file into array of strings

write string

write line

flush file handle

file handle position

redirect stdout to file

Directories

working directory

How to get and set the working directory.

Processes and Environment

command line arguments

How to get the command line arguments.

environment variables

How to get and set and environment variable.

Libraries and Namespaces

load library

How to load a library.

list loaded libraries

Show the list of libraries which have been loaded.

library search path

The list of directories the interpreter will search looking for a library to load.

source file

How to source a file.

r:

When sourcing a file, the suffix if any must be specified, unlike when loading library. Also, a library may contain a shared object, but a sourced file must consist of just R source code.

install package

How to install a package.

list installed packages

How to list the packages which have been installed.

Reflection

data type

How to get the data type of a value.

r:

For vectors `class` returns the *mode* of the vector which is the type of data contained in it. The possible modes are

- numeric
- complex

- logical
- character
- raw

Some of the more common class types for non-vector entities are:

- matrix
- array
- list
- factor
- data.frame

attributes

How to get the attributes for an object.

r:

Arrays and vectors do not have attributes.

methods

How to get the methods for an object.

variables in scope

How to list the variables in scope.

undefine variable

How to undefine a variable.

undefine all variables

How to undefine all variables.

eval

How to interpret a string as source code and execute it.

function documentation

How to get the documentation for a function.

list library functions

How to list the functions and other definitions in a library.

search documentation

How to search the documentation by keyword.

Debugging

benchmark code

How to benchmark code.

MATLAB

Octave Manual
MATLAB Documentation
Differences between Octave and MATLAB
Octave-Forge Packages

The basic data type of MATLAB is a matrix of floats. There is no distinction between a scalar and a 1x1 matrix, and functions that work on scalars typically work on matrices as well by performing the scalar function on each entry in the matrix and returning the results in a matrix with the same dimensions. Operators such as the logical operators ('&' '|' '!'), relational operators ('==', '!=', '<', '>'), and arithmetic operators ('+', '-') all work this way. However the multiplication '*' and division '/' operators perform matrix multiplication and matrix division, respectively. The `.*` and `./` operators are available if entry-wise multiplication or division is desired.

Floats are by default double precision; single precision can be specified with the *single* constructor. MATLAB has convenient matrix literal notation: commas or spaces can be used to separate row entries, and semicolons or newlines can be used to separate rows.

Arrays and vectors are implemented as single-row (`1xn`) matrices. As a result an *n*-element vector must be transposed before it can be multiplied on the right of a `mxn` matrix.

Numeric literals that lack a decimal point such as *17* and *-34* create floats, in contrast to most other programming languages. To create an integer, an integer constructor which specifies the size such as *int8* and *uint16* must be used. Matrices of integers are supported, but the entries in a given matrix must all have the same numeric type.

Strings are implemented as single-row (`1xn`) matrices of characters. Matrices cannot contain strings. If a string is put in matrix literal, each character in the string becomes an entry in the resulting matrix. This is consistent with how matrices are treated if they are nested inside another matrix. The following literals all yield the same string or `1xn` matrix of characters:

```
'foo'
[ 'f' 'o' 'o' ]
[ 'foo' ]
[ [ 'f' 'o' 'o' ] ]
```

true and *false* are functions which return matrices of ones and zeros. The ones and zeros have type *logical* instead of *double*, which is created by the literals 1 and 0. Other than having a different class, the 0 and 1 of type *logical* behave the same as the 0 and 1 of type *double*.

MATLAB has a tuple type (in MATLAB terminology, a cell array) which can be used to hold multiple strings. It can also hold values with different types.

R

An Introduction to R
Advanced R Programming
The Comprehensive R Archive Network

The primitive data types of R are vectors of floats, vectors of strings, and vectors of booleans. There is no distinction between a scalar and a vector with one entry in it. Functions and operators which accept a scalar argument will typically accept a vector argument, returning a vector of the same size with the scalar operation performed on each the entries of the original vector.

The scalars in a vector must all be of the same type, but R also provides a *list* data type which can be used as a tuple (entries accessed by index), record (entries accessed by name), or even as a dictionary.

In addition R provides a *data frame* type which is a list (in R terminology) of vectors all of the same length. Data frames are equivalent to the data sets of other statistical analysis packages.

NumPy

NumPy and SciPy Documentation
matplotlib intro
NumPy for Matlab Users
Pandas Documentation
Pandas Method/Attribute Index

NumPy is a Python library which provides a data type called `array`. It differs from the Python `list` data type in the following ways:

- N-dimensional. Although the `list` type can be nested to hold higher dimension data, the `array` can hold higher dimension data in a space efficient manner without using indirection.
- homogeneous. The elements of an `array` are restricted to be of a specified type. The NumPy library introduces new primitive types not available in vanilla Python. However, the element type of an array can be `object` which permits storing anything in the array.

In the reference sheet the array section covers the vanilla Python `list` and the multidimensional array section covers the NumPy `array`.

List the NumPy primitive types

SciPy, Matplotlib, and Pandas are libraries which depend on Numpy.

Julia

http://julialang.org/