Hyperpolyglot

Interpreted Languages: JavaScript, PHP, Python, Ruby (Sheet One)

a side-by-side reference sheet

	javascript	php	python	ruby
versions used		*5.4; 5.5*	*2.7; 3.3*	*1.9; 2.0*

show version	`$ node --version`	`$ php --version`	`$ python -V`
			`$ python --version`
				`$ ruby --version`

implicit prologue	`<script src="underscore.js">`
	`</script>`
		none	`import os, re, sys`	*none*

grammar and invocation				
	javascript	**php**	**python**	**ruby**
interpreter	`$ node foo.js`	`$ php -f foo.php`	`$ python foo.py`	`$ ruby foo.rb`
repl	`$ node`	`$ php -a`	`$ python`	`$ irb`

command line program	`$ node -e 'var sys = require("sys");`
	`sys.puts("hi world!");'`
		`$ php -r 'echo "hi\n";'`	`$ python -c 'print("hi")'`	`$ ruby -e 'puts "hi"'`

block delimiters	`{}`	`{}`	`:` *and offside rule*	`{}`
				`do end`

statement separator	*; or newline* *newline not separator inside (), [], {}, "", '', or after binary operator* *newline sometimes not separator when following line would not parse as a valid statement*	*;* *statements must be semicolon terminated inside {}*	*newline or ;* *newlines not separators inside (), [], {}, triple quote literals, or after backslash: \*	*newline or ;* *newlines not separators inside (), [], {}, \`\`, '', "", or after binary operator or backslash: \*
source code encoding		*none*	*Python 3 source is UTF-8 by default* `# -*- coding: utf-8 -*-`	*Ruby 2.0 source is UTF-8 by default* `# -*- coding: utf-8 -*-`

end-of-line comment	`// comment`	`// comment`
		`# comment`
			`# comment`	`# comment`

multiple line comment	`/* line`
	`another line */`
		`/* comment line`
		`another line */`
			use triple quote string literal:
			
			`'''comment line`
			`another line'''`
				`=begin`
				`comment line`
				`another line`
				`=end`

variables and expressions				
	javascript	**php**	**python**	**ruby**

local variable	`var x = 1;`	`# in function body:`
		`$v = NULL;`
		`$a = [];`
		`$d = [];`
			`# in function body:`
			`v = None`
			`a, d = [], {}`
			`x = 1`
				`v = nil`
				`a, d = [], {}`
				`x = 1`

		`$x = 1;`
		`list($y, $z) = [2, 3];`
			`y, z = 2, 3`	`y, z = 2, 3`

regions which define lexical scope	*top level:* *html page* *nestable:* *function*	*top level:* *function or method body* *nestable (with use clause):* *anonymous function body*	*nestable (read only):* *function or method body*	*top level:* *file* *class block* *module block* *method body* *nestable:* *anonymous function body* *anonymous block*

global variable	`// assign without using var`
	`g = 1;`
	
	`function incr_global () { g++; }`
		`list($g1, $g2) = [7, 8];`
		
		`function swap_globals() {`
		`  global $g1, $g2;`
		`  list($g1, $g2) = [$g2, $g1];`
		`}`
			`g1, g2 = 7, 8`
			
			`def swap_globals():`
			`  global g1, g2`
			`  g1, g2 = g2, g1`
				`$g1, $g2 = 7, 8`
				
				`def swap_globals`
				`  $g1, $g2 = $g2, $g1`
				`end`

constant	*none*	`define("PI", 3.14);`	`# uppercase identifiers`
			`# constant by convention`
			`PI = 3.14`
				`# warning if capitalized`
				`# identifier is reassigned`
				`PI = 3.14`

assignment	`x = 1;`	`$v = 1;`	*assignments can be chained but otherwise don't return values:* `v = 1`	`v = 1`

parallel assignment	*none*	`list($x, $y, $z) = [1 ,2, 3];`
		
		`# 3 is discarded:`
		`list($x, $y) = [1, 2, 3];`
		
		`# $z set to NULL:`
		`list($x, $y, $z) = [1, 2];`
			`x, y, z = 1, 2, 3`
			
			`# raises ValueError:`
			`x, y = 1, 2, 3`
			
			`# raises ValueError:`
			`x, y, z = 1, 2`
				`x, y, z = 1, 2, 3`
				
				`# 3 is discarded:`
				`x, y = 1, 2, 3`
				
				`# z set to nil:`
				`x, y, z = 1, 2`

swap	`tmp = x;`
	`x = y;`
	`y = tmp;`
		`list($x, $y) = [$y, $x];`	`x, y = y, x`	`x, y = y, x`

compound assignment
arithmetic, string, logical, bit
	`+= -= *= /=` *none* `%=`
	`+=`
	none
	`<<= >>= &= |= ^=`
		`+= -= *=` *none* `/= %= **=`
		`.=` *none*
		`&= |=` *none*
		`<<= >>= &= |= ^=`
			`# do not return values:`
			`+= -= *= /= //= %= **=`
			`+= *=`
			`&= |= ^=`
			`<<= >>= &= |= ^=`
				`+= -= *= /=` *none* `%= **=`
				`+= *=`
				`&&= ||= ^=`
				`<<= >>= &= |= ^=`

increment and decrement	`var x = 1;`
	`var y = ++x;`
	`var z = --y;`
		`$x = 1;`
		`$y = ++$x;`
		`$z = --$y;`
			none	`x = 1`
				`# x and y not mutated:`
				`y = x.succ`
				`z = y.pred`

null	`null`	`NULL # case insensitive`	`None`	`nil`

null test	`v === null`	`is_null($v)`
		`! isset($v)`
			`v == None`
			`v is None`
				`v == nil`
				`v.nil?`

undefined variable access	`undefined`	`NULL`	*raises* `NameError`	*raises* `NameError`
conditional expression	`x > 0 ? x : -x`	`$x > 0 ? $x : -$x`	`x if x > 0 else -x`	`x > 0 ? x : -x`
arithmetic and logic				
	javascript	**php**	**python**	**ruby**
true and false	`true false`	`TRUE FALSE # case insensitive`	`True False`	`true false`
falsehoods	`false null undefined "" 0 NaN`	`FALSE NULL 0 0.0 "" "0" []`	`False None 0 0.0 '' [] {}`	`false nil`

logical operators	`&& || !`	`&& || !`
		lower precedence:
		`and or xor`
			`and or not`	`&& || !`
				lower precedence:
				`and or not`

relational operators	`=== !== < > >= <=`
	
	perform type coercion:
	`== !=`
		`== !=` *or* `<> > < >= <=`
		no conversion: `=== !==`
			relational operators are chainable:
			`== != > < >= <=`
				`== != > < >= <=`

min and max	`Math.min(1, 2, 3)`
	`Math.max(1, 2, 3)`
	
	`Math.min.apply(Math, [1, 2, 3])`
	`Math.max.apply(Math, [1, 2, 3])`
		`min(1, 2, 3)`
		`max(1, 2, 3)`
		`$a = [1, 2, 3]`
		`min($a)`
		`max($a)`
			`min(1, 2, 3)`
			`max(1, 2, 3)`
			
			`min([1, 2, 3])`
			`max([1, 2, 3])`
				`[1, 2, 3].min`
				`[1, 2, 3].max`

three value comparison	*none*	*none*	*removed from Python 3:*
			`cmp(0, 1)`
			`cmp('do', 're')`
				`0 <=> 1`
				`"do" <=> "re"`

arithmetic operators *addition, subtraction, multiplication, float division, quotient, remainder*	`+ - * /` *none* `%`	`+ - * /` *none* `%`	`+ - *` *see note* `// %` *Python 2 does not have an operator which performs float division on integers. In Python 3* `/` *always performs float division.*	`+ - * x.fdiv(y) / %`
integer division	`Math.floor(x / y)`	`(int)(13 / 5)`	`13 // 5`	`13 / 5`
divmod	*none*	*none*	`q, r = divmod(13, 5)`	`q, r = 13.divmod(5)`
integer division by zero	*returns assignable value Infinity, NaN, or -Infinity depending upon whether dividend is positive, zero, or negative.* *There are literals for Infinity and NaN.*	*returns* `FALSE` *with warning*	*raises* `ZeroDivisionError`	*raises* `ZeroDivisionError`

float division	`13 / 5`	`13 / 5`	`float(13) / 5`
			
			`# Python 3:`
			`13 / 5`
				`13.to_f / 5` *or*
				`13.fdiv(5)`

float division by zero	*same behavior as for integers*	*returns* `FALSE` *with warning*	*raises* `ZeroDivisionError`	*returns* `-Infinity, NaN,` *or* `Infinity`
power	`Math.pow(2, 32)`	`pow(2, 32)`	`2**32`	`2**32`

sqrt	`Math.sqrt(2)`	`sqrt(2)`	`import math`
			
			`math.sqrt(2)`
				`include Math`
				
				`sqrt(2)`

sqrt -1	`NaN`	`NaN`	`# raises ValueError:`
			`import math`
			`math.sqrt(-1)`
			
			`# returns complex float:`
			`import cmath`
			`cmath.sqrt(-1)`
				raises `Errno::EDOM`

transcendental functions	`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`	`from math import exp, log, \`
			`sin, cos, tan, asin, acos, atan, atan2`
				`include Math`
				
				`exp log sin cos tan asin acos atan atan2`

transcendental constants	`Math.PI`	`M_PI M_E`	`import math`	`include Math`

π and e	`Math.E`		`math.pi math.e`	`PI E`

float truncation	*none*
	`Math.round(3.1)`
	`Math.floor(3.1)`
	`Math.ceil(3.1)`
		`(int)$x`
		`round($x)`
		`ceil($x)`
		`floor($x)`
			`import math`
			
			`int(x)`
			`int(round(x))`
			`math.ceil(x)`
			`math.floor(x)`
				`x.to_i`
				`x.round`
				`x.ceil`
				`x.floor`

absolute value	`Math.abs(-3)`	`abs($x)`	`abs(x)`	`x.abs`
integer overflow	*all numbers are floats*	*converted to float*	*becomes arbitrary length integer of type* `long`	*becomes arbitrary length integer of type* `Bignum`
float overflow	`Infinity`	`INF`	*raises* `OverflowError`	`Infinity`

rational construction		*none*	`from fractions import Fraction`
			
			`x = Fraction(22, 7)`
				`require 'rational'`
				
				`x = Rational(22, 7)`

rational decomposition		*none*	`x.numerator` `x.denominator`	`x.numerator` `x.denominator`

complex construction		*none*	`z = 1 + 1.414j`	`require 'complex'`
				
				`z = 1 + 1.414.im`

complex decomposition
real and imaginary component, argument, absolute value, conjugate
		none	`import cmath`
			
			`z.real`
			`z.imag`
			`cmath.phase(z)`
			`abs(z)`
			`z.conjugate()`
				`z.real`
				`z.imag`
				`z.arg`
				`z.abs`
				`z.conj`

random number
uniform integer, uniform float, normal float
	`Math.floor(Math.random() * 100)`
	`Math.random()`
	none
		`rand(0,99)`
		`lcg_value()`
		none
			`import random`
			
			`random.randint(0, 99)`
			`random.random()`
			`random.gauss(0, 1)`
				`rand(100)`
				`rand`
				none

random seed
set, get, restore
	none	`srand(17);`
		
		none
			`import random`
			
			`random.seed(17)`
			`seed = random.getstate()`
			`random.setstate(seed)`
				`srand(17)`
				
				`seed = srand`
				`srand(seed)`

bit operators	`<< >> & | ^ ~`	`<< >> & | ^ ~`	`<< >> & | ^ ~`	`<< >> & | ^ ~`

binary, octal, and hex literals	*none*
	`052 // deprecated`
	`0x2a`
		`0b101010`
		`052`
		`0x2a`
			`0b101010`
			`052`
			`0x2a`
				`0b101010`
				`052`
				`0x2a`

radix
convert integer to and from string with radix
	`(42).toString(7)`
	??
		`base_convert("42", 10, 7);`
		`base_convert("60", 7, 10);`
			none
			`int("60", 7)`
				`42.to_s(7)`
				`"60".to_i(7)`

strings				
	javascript	**php**	**python**	**ruby**
string type	`String`	`string`	`str`	`String`

string literal	"don't say \"no\"" 'don\'t say "no"'	"don't say \"no\"" 'don\'t say "no"'	'don\'t say "no"' "don't say \"no\"" "don't " 'say "no"' '''don't say "no"''' """don't say "no\""""	"don't say \"no\"" 'don\'t say "no"' "don't " 'say "no"'
newline in literal	*yes*	'first line second line' "first line second line"	*triple quote literals only:* '''first line second line''' """first line second line"""	'first line second line' "first line second line"
literal escapes	*single and double quotes:* \b \f \n \r \t \v \uhhhh \xhh \" \' \\	*double quoted:* \f \n \r \t \v \x*hh* \$ \" \\ *ooo* *single quoted:* \' \\	*single and double quoted:* *newline* \\ \' \" \a \b \f \n \r \t \v *ooo* \x*hh* *Python 3:* \u*hhhh* \U*hhhhhhhh*	*double quoted:* \a \b \c*x* \e \f \n \r \s \t \v \x*hh* *ooo* \u*hhhh* \u{*hhhhh*} *single quoted:* \' \\

here document	*none*	$word = "amet";
		
		$s = <<<EOF
		lorem ipsum
		dolor sit $word
		EOF;
			none	word = "amet"
				
				s = <<EOF
				lorem ipsum
				dolor sit #{word}
				EOF

variable interpolation	// None; use string concatenation.
	// Both of these expressions are '11':
	1 + "1"
	"1" + 1
		$count = 3;
		$item = "ball";
		echo "$count ${item}s\n";
			count = 3
			item = 'ball'
			print('{count} {item}s'.format(
			  **locals()))
				count = 3
				item = "ball"
				puts "#{count} #{item}s"

expression interpolation	*none*	*none*	'1 + 1 = {}'.format(1 + 1)	"1 + 1 = #{1 + 1}"

format	// None; use string concatenation.
	// Evaluates to "12.35":
	12.3456.toFixed(2)
		$fmt = "lorem %s %d %f";
		sprintf($fmt, "ipsum", 13, 3.7);
			'lorem %s %d %f' % ('ipsum', 13, 3.7)
			
			fmt = 'lorem {0} {1} {2}'
			fmt.format('ipsum', 13, 3.7)
				"lorem %s %d %f" % ["ipsum", 13, 3.7]

are strings mutable?	*no*	$s = "bar";
		$s2 = $s;
		# sets s to "baz"; s2 is unchanged:
		$s[2] = "z";
			no	s = "bar"
				s2 = s
				# sets s and s2 to "baz":
				s[2] = "z"

copy string	*none*	$s2 = $s;	*none*	s = "bar"
				s2 = s.clone
				# s2 is not altered:
				s[2] = "z"

concatenate	s = "Hello, " + "World!";	$s = "Hello, ";
		$s2 = $s . "World!";
			s = 'Hello, '
			s2 = s + 'World!'
			
			# juxtaposition can be used to
			# concatenate literals:
			s2 = 'Hello, ' "World!"
				s = "Hello, "
				s2 = s + "World!"
				
				# juxtaposition can be used to
				# concatenate literals:
				s2 ="Hello, " 'World!'

replicate	var hbar = Array(80).join("-");	$hbar = str_repeat("-", 80);	hbar = '-' * 80	hbar = "-" * 80

translate case
to upper, to lower
	"lorem".toUpperCase()
	"LOREM".toLowerCase()
		strtoupper("lorem")
		strtolower("LOREM")
			'lorem'.upper()
			'LOREM'.lower()
				"lorem".upcase
				"LOREM".downcase

capitalize
string, words
	none	ucfirst("lorem")
		ucwords("lorem ipsum")
			import string
			
			'lorem'.capitalize()
			string.capwords('lorem ipsum')
				"lorem".capitalize
				none

trim
both sides, left, right
	" lorem ".trim()
	# some browsers:
	" lorem".trimLeft()
	"lorem ".trimRight()
		trim(" lorem ")
		ltrim(" lorem")
		rtrim("lorem ")
			' lorem '.strip()
			' lorem'.lstrip()
			'lorem '.rstrip()
				" lorem ".strip
				" lorem".lstrip
				"lorem ".rstrip

pad
on right, on left, centered
	none	str_pad("lorem", 10)
		str_pad("lorem", 10, " ", STR_PAD_LEFT)
		str_pad("lorem", 10, " ", STR_PAD_BOTH)
			'lorem'.ljust(10)
			'lorem'.rjust(10)
			'lorem'.center(10)
				"lorem".ljust(10)
				"lorem".rjust(10)
				"lorem".center(10)

number to string	"value: " + 8	"value: " . 8	'value: ' + str(8)	"value: " + 8.to_s
string to number	7 + parseInt("12", 10) 73.9 + parseFloat(".037")	7 + "12" 73.9 + ".037"	7 + int('12') 73.9 + float('.037')	7 + "12".to_i 73.9 + ".037".to_f
join	["do", "re", "mi"].join(" ")	$a = ["do", "re", "mi", "fa"]; implode(" ", $a)	' '.join(['do', 're', 'mi', 'fa']) # raises TypeError: ' '.join([1, 2, 3])	%w(do re mi fa).join(' ') # implicitly converted to strings: [1, 2, 3].join(' ')
split	"do re mi".split(" ")	explode(" ", "do re mi fa")	'do re mi fa'.split()	"do re mi fa".split
split in two	*none*	preg_split('/\s+/', "do re mi fa", 2)	'do re mi fa'.split(None, 1)	"do re mi fa".split(/\s+/, 2)
split and keep delimiters	*none*	preg_split('/(\s+)/', "do re mi fa", NULL, PREG_SPLIT_DELIM_CAPTURE)	re.split('(\s+)', 'do re mi fa')	"do re mi fa".split(/(\s+)/)
length	"lorem".length	strlen("lorem")	len('lorem')	"lorem".length "lorem".size

index of substring
first, last
	"lorem ipsum".indexOf("ipsum")	# returns FALSE if not found:
		strpos("do re re", "re")
		strrpos("do re re", "re")
			# raises ValueError if not found:
			'do re re'.index('re')
			'do re re'.rindex('re')
			
			# returns -1 if not found:
			'do re re'.find('re')
			'do re re'.rfind('re')
				# returns nil if not found:
				"do re re".index("re")
				"do re re".rindex("re")

extract substring *by start and length, by start and end, by successive starts*	"lorem ipsum".substr(6, 5) "lorem ipsum".substring(6, 11)	substr("lorem ipsum", 6, 5) *none* *none*	*none* *none* 'lorem ipsum'[6:11]	"lorem ipsum"[6, 5] "lorem ipsum"[6..10] "lorem ipsum"[6...11]
extract character	"lorem ipsum"[6]	# syntax error to use index notation # directly on string literal: $s = "lorem ipsum"; $s[6];	'lorem ipsum'[6]	"lorem ipsum"[6]

chr and ord	String.fromCharCode(65)
	"A".charCodeAt(0)
		chr(65)
		ord("A")
			chr(65)
			ord('A')
				65.chr
				"A".ord

to array of characters	"abcd".split("")	str_split("abcd")	list('abcd')	"abcd".split("")

translate characters	*none*	$ins = implode(range("a", "z"));
		$outs = substr($ins, 13, 13) .
		  substr($ins, 0, 13);
		strtr("hello", $ins, $outs)
			from string import lowercase as ins
			from string import maketrans
			
			outs = ins[13:] + ins[:13]
			'hello'.translate(maketrans(ins,outs))
				"hello".tr("a-z", "n-za-m")

delete characters	*none*	$vowels = str_split("aeiou"); $s = "disemvowel me"; $s = str_replace($vowels, "", $s);	"disemvowel me".translate(None, "aeiou")	"disemvowel me".delete("aeiou")
squeeze characters	*none*	$s = "too much space"; $s = = preg_replace('/(\s)+/', '\1', $s);	re.sub('(\s)+', r'\1', 'too much space')	"too much space".squeeze(" ")
regular expressions				
	javascript	**php**	**python**	**ruby**
literal, custom	/lorem|ipsum/	'/lorem|ipsum/'	re.compile('lorem|ipsum')	/lorem|ipsum/

delimited literal		'(/etc/hosts)'	none	%r(/etc/hosts)
character class abbreviations	. \d \D \s \S \w \W	. \d \D \h \H \s \S \v \V \w \W	. \d \D \s \S \w \W	. \d \D \h \H \s \S \w \W
anchors	^ $ \b \B	^ $ \A \b \B \z \Z	^ $ \A \b \B \Z	^ $ \A \b \B \z \Z

match test	if (s.match(/1999/)) {
	  alert("party!");
	}
		if (preg_match('/1999/', $s)) {
		  echo "party!\n";
		}
			if re.search('1999', s):
			  print('party!')
				if /1999/.match(s)
				  puts "party!"
				end

case insensitive match test	"Lorem".match(/lorem/i)	preg_match('/lorem/i', "Lorem")	re.search('lorem', 'Lorem', re.I)	/lorem/i.match("Lorem")
modifiers	g i m	e i m s x	re.I re.M re.S re.X	i o m x
substitution	s = "do re mi mi mi"; s.replace(/mi/g, "ma");	$s = "do re mi mi mi"; $s = preg_replace('/mi/', "ma", $s);	s = 'do re mi mi mi' s = re.compile('mi').sub('ma', s)	s = "do re mi mi mi" s.gsub!(/mi/, "ma")

match, prematch, postmatch	m = /\d{4}/.exec(s);
	if (m) {
	  match = m[0];
	  # no prematch or postmatch
	}
		none	m = re.search('\d{4}', s)
			if m:
			  match = m.group()
			  prematch = s[0:m.start(0)]
			  postmatch = s[m.end(0):len(s)]
				m = /\d{4}/.match(s)
				if m
				  match = m[0]
				  prematch = m.pre_match
				  postmatch = m.post_match
				end

group capture	rx = /^(\d{4})-(\d{2})-(\d{2})$/;
	m = rx.exec('2009-06-03');
	yr = m[1];
	mo = m[2];
	dy = m[3];
		$s = "2010-06-03";
		$rx = '/(\d{4})-(\d{2})-(\d{2})/';
		preg_match($rx, $s, $m);
		list($_, $yr, $mo, $dy) = $m;
			rx = '(\d{4})-(\d{2})-(\d{2})'
			m = re.search(rx, '2010-06-03')
			yr, mo, dy = m.groups()
				rx = /(\d{4})-(\d{2})-(\d{2})/
				m = rx.match("2010-06-03")
				yr, mo, dy = m[1..3]

named group capture	none	$s = "foo.txt";
		$rx = '/^(?P<file>.+)\.(?
		P<suffix>.+)$/';
		preg_match($rx, $s, $m);
		
		$m["file"]
		$m["suffix"]
			rx = '^(?P<file>.+)\.(?P<suffix>.+)$'
			m = re.search(rx, ''foo.txt')
			
			m.groupdict()['file']
			m.groupdict()['suffix']
				rx = /^(?<file>.+)\.(?<suffix>.+)$/
				m = rx.match('foo.txt')
				
				m["file"]
				m["suffix"]

scan	var a = "dolor sit amet".match(/\w+/g);	$s = "dolor sit amet";
		preg_match_all('/\w+/', $s, $m);
		$a = $m[0];
			s = 'dolor sit amet'
			a = re.findall('\w+', s)
				a = "dolor sit amet".scan(/\w+/)

backreference in match and substitution	/(\w+) \1/.exec("do do")
	
	"do re".replace(/(\w+) (\w+)/, '$2 $1')
		preg_match('/(\w+) \1/', "do do")
		
		$s = "do re";
		$rx = '/(\w+) (\w+)/';
		$s = preg_replace($rx, '\2 \1', $s);
			none
			
			rx = re.compile('(\w+) (\w+)')
			rx.sub(r'\2 \1', 'do re')
				/(\w+) \1/.match("do do")
				
				"do re".sub(/(\w+) (\w+)/, '\2 \1')

recursive regex	none	'/\(([^()]*|($R))\)/'	none	/(?<foo>\(([^()]*|\g<foo>)*\))/

dates and time

	javascript	php	python	ruby
date/time type	Date	DateTime	datetime.datetime	Time

current date/time	var t = new Date();	$t = new DateTime("now");
		$utc_tmz = new DateTimeZone("UTC");
		$utc = new DateTime("now", $utc_tmz);
			import datetime
			
			t = datetime.datetime.now()
			utc = datetime.datetime.utcnow()
				t = Time.now
				utc = Time.now.utc

to unix epoch, from unix epoch	Math.round(t.getTime() / 1000)
	var epoch = 1315716177;
	var t2 = new Date(epoch * 1000);
		$epoch = $t->getTimestamp();
		$t2 = new DateTime();
		$t2->setTimestamp(1304442000);
			from datetime import datetime as dt
			
			epoch = int(t.strftime("%s"))
			t2 = dt.fromtimestamp(1304442000)
				epoch = t.to_i
				t2 = Time.at(1304442000)

			import datetime	

current unix epoch	(new Date()).getTime() / 1000	$epoch = time();	t = datetime.datetime.now()
			epoch = int(t.strftime("%s"))
				epoch = Time.now.to_i

strftime	*none*	strftime("%Y-%m-%d %H:%M:%S", $epoch);
		date("Y-m-d H:i:s", $epoch);
		$t->format("Y-m-d H:i:s");
			t.strftime('%Y-%m-%d %H:%M:%S')	t.strftime("%Y-%m-%d %H:%M:%S")

default format example	Tue Apr 01 2014 13:05:36 GMT-0700 (PDT)	*no default string representation*	2011-08-23 19:35:59.411135	2011-08-23 17:44:53 -0700

strptime	*none*	$fmt = "Y-m-d H:i:s";
		$s = "2011-05-03 10:00:00";
		$t = DateTime::createFromFormat($fmt,
		  $s);
			from datetime import datetime
			
			s = '2011-05-03 10:00:00'
			fmt = '%Y-%m-%d %H:%M:%S'
			t = datetime.strptime(s, fmt)
				require 'date'
				
				s = "2011-05-03 10:00:00"
				fmt = "%Y-%m-%d %H:%M:%S"
				t = Date.strptime(s, fmt).to_time

parse date w/o format	var t = new Date("July 7, 1999");	$epoch = strtotime("July 7, 1999");	# pip install python-dateutil
			import dateutil.parser
			
			s = 'July 7, 1999'
			t = dateutil.parser.parse(s)
				require 'date'
				
				s = "July 7, 1999"
				t = Date.parse(s).to_time

get date parts	t.getFullYear() t.getMonth() + 1 t.getDate() # getDay() is day of week	(int)$t->format("Y") (int)$t->format("m") (int)$t->format("d")	t.year t.month t.day	t.year t.month t.day

get time parts	t.getHours()
	t.getMinutes()
	t.getSeconds()
		(int)$t->format("H")
		(int)$t->format("i")
		(int)$t->format("s")
			t.hour
			t.minute
			t.second
				t.hour
				t.min
				t.sec

build date/time from parts	var yr = 1999;
	var mo = 9;
	var dy = 10;
	var hr = 23;
	var mi = 30;
	var ss = 0;
	var t = new Date(yr,mo-1,dy,hr,mi,ss);
			import datetime
			
			yr = 1999
			mo = 9
			dy = 10
			hr = 23
			mi = 30
			ss = 0
			t = datetime.datetime(yr,mo,dy,hr,mi,ss)
				yr = 1999
				mo = 9
				dy = 10
				hr = 23
				mi = 30
				ss = 0
				t = Time.new(yr,mo,dy,hr,mi,ss)

result of date subtraction	number *containing time difference in milliseconds*	DateInterval *object if* diff *method used:* $fmt = "Y-m-d H:i:s"; $s = "2011-05-03 10:00:00"; $then = DateTime::createFromFormat($fmt, $s); $now = new DateTime("now"); $interval = $now->diff($then);	datetime.timedelta *object* *use* total_seconds() *method to convert to float representing difference in seconds*	Float *containing time difference in seconds*

add time duration	var t1 = new Date();
	var delta = (10 * 60 + 3) * 1000;
	var t2 = new Date(t1.getTime() + delta);
		$now = new DateTime("now");
		$now->add(new DateInterval("PT10M3S");
			import datetime
			
			delta = datetime.timedelta(
			  minutes=10,
			  seconds=3)
			t = datetime.datetime.now() + delta
				require 'date/delta'
				
				s = "10 min, 3 s"
				delta = Date::Delta.parse(s).in_secs
				t = Time.now + delta

local timezone		*DateTime objects can be instantiated without specifying the timezone if a default is set:* $s = "America/Los_Angeles"; date_default_timezone_set($s);	*a datetime object has no timezone information unless a tzinfo object is provided when it is created*	*if no timezone is specified the local timezone is used*

arbitrary timezone			# pip install pytz
			import pytz
			import datetime
			
			tmz = pytz.timezone('Asia/Tokyo')
			utc = datetime.datetime.utcnow()
			utc_dt = datetime.datetime(
			  *utc.timetuple()[0:5],
			  tzinfo=pytz.utc)
			jp_dt = utc_dt.astimezone(tmz)
				# gem install tzinfo
				require 'tzinfo'
				
				tmz = TZInfo::Timezone.get("Asia/Tokyo")
				jp_time = tmz.utc_to_local(Time.now.utc)

			import time	

timezone name; offset from UTC; is daylight savings?		$tmz = date_timezone_get($t);
		timezone_name_get($tmz);
		date_offset_get($t) / 3600;
		$t->format("I");
			tm = time.localtime()
			
			time.tzname[tm.tm_isdst]
			(time.timezone / -3600) + tm.tm_isdst
			tm.tm_isdst
				t.zone
				t.utc_offset / 3600
				t.dst?

microseconds		list($frac, $sec) = explode(" ", microtime()); $usec = $frac * 1000 * 1000;	t.microsecond	t.usec

sleep	*none*	# a float argument will be truncated
		# to an integer:
		sleep(1);
			import time
			
			time.sleep(0.5)
				sleep(0.5)

timeout		*use* set_time_limit *to limit execution time of the entire script; use* stream_set_timeout *to limit time spent reading from a stream opened with* fopen *or* fsockopen	import signal, time
			
			class Timeout(Exception): pass
			
			def timeout_handler(signo, fm):
			  raise Timeout()
			
			signal.signal(signal.SIGALRM,
			  timeout_handler)
			
			try:
			  signal.alarm(5)
			  time.sleep(10)
			except Timeout:
			  pass
			signal.alarm(0)
				require 'timeout'
				
				begin
				  Timeout.timeout(5) do
				    sleep(10)
				  end
				rescue Timeout::Error
				end

arrays

	javascript	php	python	ruby

literal	a = [1, 2, 3, 4]	$a = [1, 2, 3, 4];
		
		# older syntax:
		$a = array(1, 2, 3, 4);
			a = [1, 2, 3, 4]	a = [1, 2, 3, 4]
				
				# a = ['do', 're', 'mi']
				a = %w(do re mi)

size	a.length	count($a)	len(a)	a.size a.length # same as size

empty test	// TypeError if a is null or undefined:
	a.length === 0
		# NULL tests as empty:
		!$a
			# None tests as empty:
			not a
				# NoMethodError if a is nil:
				a.empty?

lookup	a[0]	$a[0] # PHP uses the same type for arrays and # dictionaries; indices can be negative # integers or strings	a[0] # returns last element: a[-1]	a[0] # returns last element: a[-1]
update	a[0] = "lorem"	$a[0] = "lorem";	a[0] = 'lorem'	a[0] = "lorem"

out-of-bounds behavior	*returns* undefined	$a = [];
		# evaluates as NULL:
		$a[10];
		# increases array size to one:
		$a[10] = "lorem";
			a = []
			# raises IndexError:
			a[10]
			# raises IndexError:
			a[10] = 'lorem'
				a = []
				# evaluates as nil:
				a[10]
				# increases array size to 11:
				a[10] = "lorem"

index of element	[6, 7, 7, 8].indexOf(7)
	[6, 7, 7, 8].lastIndexOf(7)
	// returns -1 if not found
		$a = ["x", "y", "z", "w"];
		$i = array_search("y", $a);
			a = ['x', 'y', 'z', 'w']
			i = a.index('y')
				a = %w(x y z w)
				i = a.index("y")

slice *by endpoints, by length*	["a", "b", "c", "d"].slice(1,3)	# select 3rd and 4th elements: *none* array_slice($a, 2, 2)	# select 3rd and 4th elements: a[2:4] a[2:2 + 2]	# select 3rd and 4th elements: a[2..3] a[2, 2]
slice to end	["a", "b", "c", "d"].slice(1)	array_slice($a, 1)	a[1:]	a[1..-1]

manipulate back	`a = [6, 7, 8];`
	`a.push(9);`
	`i = a.pop();`
		`$a = [6, 7, 8];`
		`array_push($a, 9);`
		`$a[] = 9; # same as array_push`
		`array_pop($a);`
			`a = [6, 7, 8]`
			`a.append(9)`
			`a.pop()`
				`a = [6, 7, 8]`
				`a.push(9)`
				`a << 9 # same as push`
				`a.pop`

manipulate front	`a = [6, 7, 8];`
	`a.unshift(5);`
	`i = a.shift();`
		`$a = [6, 7, 8];`
		`array_unshift($a, 5);`
		`array_shift($a);`
			`a = [6, 7, 8]`
			`a.insert(0, 5)`
			`a.pop(0)`
				`a = [6, 7, 8]`
				`a.unshift(5)`
				`a.shift`

concatenate	`a = [1, 2, 3].concat([4, 5, 6]);`	`$a = [1, 2, 3];`
		`$a2 = array_merge($a, [4, 5, 6]);`
		`$a = array_merge($a, [4, 5, 6]);`
			`a = [1, 2, 3]`
			`a2 = a + [4, 5, 6]`
			`a.extend([4, 5, 6])`
				`a = [1, 2, 3]`
				`a2 = a + [4, 5, 6]`
				`a.concat([4, 5, 6])`

replicate	*none*		`a = [None] * 10`
			`a = [None for i in range(0, 10)]`
				`a = [nil] * 10`
				`a = Array.new(10, nil)`

copy
address copy,
shallow copy,
deep copy
	`a = [1, 2, [3, 4]];`
	`a2 = a;`
	`a3 = a.slice(0);`
	`a4 = JSON.parse(JSON.stringify(a));`
		`$a = [1, 2, [3, 4]];`
		`$a2 =& $a;`
		none
		`$a4 = $a;`
			`import copy`
			
			`a = [1,2,[3,4]]`
			`a2 = a`
			`a3 = list(a)`
			`a4 = copy.deepcopy(a)`
				`a = [1,2,[3,4]]`
				`a2 = a`
				`a3 = a.dup`
				`a4 = Marshal.load(Marshal.dump(a))`

arrays as function arguments	*parameter contains address copy*	*parameter contains deep copy*	*parameter contains address copy*	*parameter contains address copy*

iterate over elements	`_.each([1, 2, 3], function(n) {`
	`  alert(n);`
	`})`
		`foreach ([1, 2, 3] as $i) {`
		`  echo "$i\n";`
		`}`
			`for i in [1,2,3]:`
			`  print(i)`
				`[1,2,3].each { |i| puts i }`

iterate over indices and elements	`var len = a.length;`
	`for (var i = 0; i < len; i++ ) {`
	`  alert(a[i]);`
	`}`
		`$a = ["do", "re", "mi" "fa"];`
		`foreach ($a as $i => $s) {`
		`  echo "$s at index $i\n";`
		`}`
			`a = ['do', 're', 'mi', 'fa']`
			`for i, s in enumerate(a):`
			`  print('%s at index %d' % (s, i))`
				`a = %w(do re mi fa)`
				`a.each_with_index do |s, i|`
				`  puts "#{s} at index #{i}"`
				`end`

iterate over range	*not space efficient; use C-style for loop*	*not space efficient; use C-style for loop*	`# use range() in Python 3:`
			`for i in xrange(1, 1000001):`
			`  `*code*
				`(1..1_000_000).each do |i|`
				`  `*code*
				`end`

instantiate range as array	`var a = _.range(1, 10);`	`$a = range(1, 10);`	`a = range(1, 11)`
			Python 3:
			`a = list(range(1, 11))`
				`a = (1..10).to_a`

reverse
non-destructive,
in-place
	`var a = [1, 2, 3];`
	`a.reverse();`
		`$a = [1, 2, 3];`
		
		`array_reverse($a);`
		`$a = array_reverse($a);`
			`a = [1, 2, 3]`
			
			`a[::-1]`
			`a.reverse()`
				`a = [1, 2, 3]`
				
				`a.reverse`
				`a.reverse!`

sort
non-destructive,
in-place,
custom comparision
	`var a = [3, 1, 4, 2];`
	`a.sort();`
		`$a = ["b", "A", "a", "B"];`
		
		none
		`sort($a);`
		none, but usort sorts in place
			`a = ['b', 'A', 'a', 'B']`
			
			`sorted(a)`
			`a.sort()`
			`# custom binary comparision`
			`# removed from Python 3:`
			`a.sort(key=str.lower)`
				`a = %w(b A a B)`
				
				`a.sort`
				`a.sort!`
				`a.sort do |x, y|`
				`  x.downcase <=> y.downcase`
				`end`

dedupe
non-destructive,
in-place
	`var a = [1, 2, 2, 3];`
	
	`var a2 = _.uniq(a);`
	`a = _.uniq(a);`
		`$a = [1, 2, 2, 3];`
		
		`$a2 = array_unique($a);`
		`$a = array_unique($a);`
			`a = [1, 2, 2, 3]`
			
			`a2 = list(set(a))`
			`a = list(set(a))`
				`a = [1, 2, 2, 3]`
				
				`a2 = a.uniq`
				`a.uniq!`

membership	`_.contains(a, 7)`	`in_array(7, $a)`	`7 in a`	`a.include?(7)`

intersection	`_.intersection([1, 2], [2, 3, 4])`	`$a = [1, 2];`
		`$b = [2, 3, 4]`
		`array_intersect($a, $b)`
			`{1,2} & {2,3,4}`	`[1,2] & [2,3,4]`

union	`_.union([1, 2], [2, 3, 4])`	`$a1 = [1, 2];`
		`$a2 = [2, 3, 4];`
		`array_unique(array_merge($a1, $a2))`
			`{1,2} | {2,3,4}`	`[1,2] | [2,3,4]`

relative complement, symmetric difference	`_.difference([1,2,3], [2])`
	none
		`$a1 = [1, 2, 3];`
		`$a2 = [2];`
		`array_values(array_diff($a1, $a2))`
		none
			`{1,2,3} - {2}`
			`{1,2} ^ {2,3,4}`
				`require 'set'`
				
				`[1,2,3] - [2]`
				`Set[1,2] ^ Set[2,3,4]`

map	`// callback gets 3 args:`
	`// value, index, array`
	`a.map(function(x) { return x * x })`
		`array_map(function ($x) {`
		`   return $x * $x;`
		` }, [1, 2, 3])`
			`map(lambda x: x * x, [1,2,3])`
			`# or use list comprehension:`
			`[x * x for x in [1,2,3]]`
				`[1,2,3].map { |o| o * o }`

filter	`a.filter(function(x) { return x > 1 })`	`array_filter([1, 2, 3],`
		`  function ($x) {`
		`   return $x>1;`
		`  })`
			`filter(lambda x: x > 1, [1,2,3])`
			`# or use list comprehension:`
			`[x for x in [1,2,3] if x > 1]`
				`[1,2,3].select { |o| o > 1 }`

reduce	`a.reduce(function(m, o) {`
	`    return m + o;`
	`  }, 0)`
		`array_reduce([1, 2, 3],`
		`  function($x,$y) {`
		`   return $x + $y;`
		`  }, 0)`
			`# import needed in Python 3 only`
			`from functools import reduce`
			
			`reduce(lambda x, y: x + y, [1, 2, 3], 0)`
				`[1, 2, 3].inject(0) { |m, o| m + o }`

universal and existential tests	`var a = [1, 2, 3, 4];`
	`var even = function(x) {`
	`  return x % 2 == 0;`
	`};`
	
	`a.every(even)`
	`a.some(even)`
		use array_filter	`all(i % 2 == 0 for i in [1,2,3,4])`
			`any(i % 2 == 0 for i in [1,2,3,4])`
				`[1,2,3,4].all? {|i| i.even? }`
				`[1,2,3,4].any? {|i| i.even? }`

shuffle and sample	`var a = [1, 2, 3, 4];`
	`a = _.shuffle(a);`
	`var samp = _.sample([1, 2, 3, 4], 2);`
		`$a = [1, 2, 3, 4];`
		`shuffle($a);`
		`$samp = array_rand(|[1, 2, 3, 4], 2);`
			`from random import shuffle, sample`
			
			`a = [1, 2, 3, 4]`
			`shuffle(a)`
			`samp = sample([1, 2, 3, 4], 2)`
				`[1, 2, 3, 4].shuffle!`
				`samp = [1, 2, 3, 4].sample(2)`

flatten
one level, completely
	`var a = [1, [2, [3, 4]]];`
	`var a2 = _.flatten(a, true);`
	`var a3 = _.flatten(a);`
		none	*none*	`a = [1, [2, [3, 4]]]`
				`a2 = a.flatten(1)`
				`a3 = a.flatten`

zip	`var a = _.zip(`
	`  [1, 2, 3],`
	`  ["a", "b", "c"]);`
		`$a = array_map(NULL,`
		`  [1, 2, 3],`
		`  ["a", "b", "c"]);`
			`a = zip([1,2,3], ['a', 'b', 'c'])`	`a = [1,2,3].zip(["a", "b", "c"])`

dictionaries

	javascript	php	python	ruby

literal	`d = { "t":1, "f":0 };`
	`// keys do not need to be quoted if they`
	`// are a legal JavaScript variable name`
	`//and not a reserved word`
		`$d = ["t" => 1, "f" => 0];`
		
		`# older syntax:`
		`$d = array("t" => 1, "f" => 0);`
			`d = { 't':1, 'f':0 }`	`d = { "t" => 1, "f" => 0 }`

size	`var size = 0;`
	`for (var k in d) {`
	`  if (d.hasOwnProperty(k)) size++;`
	`}`
		`count($d)`	`len(d)`	`d.size`
				`d.length # same as size`

lookup	`d.t`
	`d["t"]`
		`$d["t"]`	`d['t']`	`d["t"]`

update	`d["t"] = 2;`
	`d.t = 2;`
		`$d["t"] = 2;`	`d['t'] = 2`	`d["t"] = 2`

out-of-bounds behavior	`var d = {};`
	`// sets s to undefined:`
	`var s = d["lorem"];`
	`// adds key/value pair:`
	`d["lorem"] = "ipsum";`
		`$d = [];`
		`# sets $s to NULL:`
		`$s = $d["lorem"];`
		`# adds key/value pair:`
		`$d["lorem"] = "ipsum";`
			`d = {}`
			`# raises KeyError:`
			`s = d['lorem']`
			`# adds key/value pair:`
			`d['lorem'] = 'ipsum'`
				`d = {}`
				`# sets s to nil:`
				`s = d["lorem"]`
				`# adds key/value pair:`
				`d["lorem"] = "ipsum"`

is key present	`d.hasOwnProperty("t");`	`array_key_exists("y", $d);`	`'y' in d`	`d.has_key?("y")`

delete entry	`delete d["t"];`
	`delete d.t;`
		`$d = [1 => "t", 0 => "f"];`
		`unset($d[1]);`
			`d = {1: True, 0: False}`
			`del d[1]`
				`d = {1 => true, 0 => false}`
				`d.delete(1)`

from array of	`var a = [["a", 1], ["b", 2], ["c", 3]];`		`a = [['a', 1], ['b', 2], ['c', 3]]`	`a = [["a", 1], ["b", 2], ["c", 3]]`

pairs, from even length array	`var d = _.object(a);`
	
	none
			`d = dict(a)`
			
			`a = ['a', 1, 'b', 2, 'c', 3]`
			`d = dict(zip(a[::2], a[1::2]))`
				`d = Hash[a]`
				
				`a = ["a", 1, "b", 2, "c", 3]`
				`d = Hash[*a]`

merge	`var d1 = {"a": 1, "b": 2};`
	`var d2 = {"c": 3, "d": 4};`
	`d1 = _.extend(d1, d2);`
		`$d1 = ["a" => 1, "b" => 2];`
		`$d2 = ["b" => 3, "c" => 4];`
		`$d1 = array_merge($d1, $d2);`
			`d1 = {'a': 1, 'b': 2}`
			`d2 = {'b': 3, 'c': 4}`
			`d1.update(d2)`
				`d1 = {"a" => 1, "b" => 2}`
				`d2 = {"b" => 3, "c" => 4}`
				`d1.merge!(d2)`

invert	`var to_num = {'t': 1, 'f': 0};`
	`var to_let = _.invert(to_num);`
		`$to_num = ["t" => 1, "f" => 0];`
		`$to_let = array_flip($to_num);`
			`to_num = {'t': 1, 'f': 0}`
			`# dict comprehensions added in 2.7:`
			`to_let = {v: k for k, v`
			`  in to_num.items()}`
				`to_num = {"t" => 1, "f" => 0}`
				`to_let = to_num.invert`

iteration	`for (var k in d) {`
	`  `*`use k or d[k]`*
	`}`
		`foreach ($d as $k => $v) {`
		`  `*`code`*
		`}`
			`for k, v in d.iteritems():`
			`  `*`code`*
			
			Python 3:
			`for k, v in d.items():`
			`  `*`code`*
				`d.each do |k,v|`
				`  `*`code`*
				`end`

keys and values as arrays	`_.keys(d)`
	`_.values(d)`
		`array_keys($d)`
		`array_values($d)`
			`d.keys()`
			`d.values()`
			
			Python 3:
			`list(d.keys())`
			`list(d.values())`
				`d.keys`
				`d.values`

sort by values	`function cmp2(a, b) {`
	`  if (a[1] < b[1]) { return -1; }`
	`  if (a[1] > b[1]) { return 1; }`
	`  return 0;`
	`}`
	
	`for (p in _.pairs(d).sort(cmp2)) {`
	`  alert(p[0] + ": " + p[1]);`
	`}`
		`asort($d);`
		
		`foreach ($d as $k => $v) {`
		`  print "$k: $v\n";`
		`}`
			`from operator import itemgetter`
			
			`pairs = sorted(d.iteritems(),`
			`  key=itemgetter(1))`
			
			`for k, v in pairs:`
			`  print('{}: {}'.format(k, v))`
				`d.sort_by {|k, v| v}.each do |k, v|`
				`  puts "#{k}: #{v}"`
				`end`

default value, computed value	*none*	`$counts = [];`
		`$counts['foo'] += 1;`
		
		extend `ArrayObject` *for computed values and defaults other than zero or empty string.*
			`from collections import defaultdict`
			
			`counts = defaultdict(lambda: 0)`
			`counts['foo'] += 1`
			
			`class Factorial(dict):`
			`  def __missing__(self, k):`
			`    if k > 1:`
			`      return k * self[k-1]`
			`    else:`
			`      return 1`
			
			`factorial = Factorial()`
				`counts = Hash.new(0)`
				`counts['foo'] += 1`
				
				`factorial = Hash.new do |h,k|`
				`  k > 1 ? k * h[k-1] : 1`
				`end`

functions

	javascript	php	python	ruby

define function	`function add(x, y) {`
	`  return x+y;`
	`}`
		`function add3($x1, $x2, $x3)`
		`{`
		`  return $x1 + $x2 + $x3;`
		`}`
			`def add3(x1, x2, x3):`
			`  return x1 + x2 + x3`
				`def add3(x1, x2, x3)`
				`  x1 + x2 + x3`
				`end`
				
				`# parens are optional and customarily`
				`# omitted when defining functions`
				`# with no parameters`

invoke function	`add(1, 2)`	`add3(1, 2, 3);`
		
		`# function names are case insensitive:`
		`ADD3(1, 2, 3);`
			`add3(1, 2, 3)`	`add3(1, 2, 3)`
				
				`# parens are optional:`
				`add3 1, 2, 3`

missing argument behavior	*set to* `undefined`	*set to* `NULL` *with warning*	*raises* `TypeError` *if number of arguments doesn't match function arity*	*raises* `ArgumentError` *if number of arguments doesn't match function arity*

default argument	*none*	`function my_log($x, $base=10)`
		`{`
		`  return log($x) / log($base);`
		`}`
		
		`my_log(42);`
		`my_log(42, M_E);`
			`import math`
			
			`def my_log(x, base=10):`
			`  return math.log(x) / math.log(base)`
			
			`my_log(42)`
			`my_log(42, math.e)`
				`def my_log(x, base=10)`
				`  Math.log(x) / Math.log(base)`
				`end`
				
				`my_log(42)`
				`my_log(42, Math::E)`

variable number of arguments	*args in* `arguments[0], arguments[1], …`
	with number of args in `arguments.length`
		`function first_and_last()`
		`{`
		
		`  $arg_cnt = func_num_args();`
		
		`  if ($arg_cnt >= 1) {`
		`    $n = func_get_arg(0);`
		`    echo "first: " . $n . "\n";`
		`  }`
		
		`  if ($arg_cnt >= 2) {`
		`    $a = func_get_args();`
		`    $n = $a[$arg_cnt-1];`
		`    echo "last: " . $n . "\n";`
		`  }`
		`}`
			`def first_and_last(*a):`
			
			`  if len(a) >= 1:`
			`    print('first: ' + str(a[0]))`
			
			`  if len(a) >= 2:`
			`    print('last: ' + str(a[-1]))`
				`def first_and_last(*a)`
				
				`  if a.size >= 1`
				`    puts "first: #{a[0]}"`
				`  end`
				
				`  if a.size >= 2`
				`    puts "last: #{a[-1]}"`
				`  end`
				`end`

pass array elements as separate arguments	*none*	`$a = [1, 2, 3];`
		
		`call_user_func_array("add3", $a);`
			`a = [2, 3]`
			
			`add3(1, *a)`
			
			`# splat operator can only be used once`
			`# and must appear after other`
			`# unnamed arguments`
				`a = [2, 3]`
				
				`add3(1, *a)`
				
				`# splat operator can be used multiple`
				`# times and can appear before regular`
				`# arguments`

named parameters	*none*	*none*	`def fequal(x, y, eps=0.01):`
			`  return abs(x - y) < eps`
			
			`fequal(1.0, 1.001)`
			`fequal(1.0, 1.001, eps=0.1**10)`
				`def fequal(x, y, opts={})`
				`  eps = opts[:eps] || 0.01`
				`  (x - y).abs < eps`
				`end`
				
				`fequal(1.0, 1.001)`
				`fequal(1.0, 1.001, :eps=>0.1**10)`
				
				`# Ruby 2.0:`
				`def fequals(x, y, eps: 0.01)`
				`  (x - y).abs < eps`
				`end`
				
				`fequals(1.0, 1.001)`
				`fequals(1.0, 1.001, eps: 0.1**10)`

pass number or string by reference	*not possible*	`function impure(&$x, &$y)`
		`{`
		`  $x += 1;`
		`  $y .= "ly";`
		`}`
		
		`$n = 7;`
		`$s = "hard";`
		`impure($n, $s);`
			not possible	*not possible*

pass array or dictionary by reference	`function impure(x, y) {`
	`  x[2] = 5;`
	`  y["f"] = -1;`
	`}`
	
	`var a = [1, 2, 3];`
	`var d = {"t": 1, "f": 0};`
	`impure(a, d);`
		`function impure(&$x, &$y)`
		`{`
		`  $x[2] = 5;`
		`  $y["f"] = -1;`
		`}`
		
		`$a = [1, 2, 3];`
		`$d = ["t" => 1, "f" => 0];`
		`impure($a, $d);`
			`def impure(x, y):`
			`  x[2] = 5`
			`  y['f'] = -1`
			
			`a = [1, 2, 3]`
			`d = {'t':1, 'f':0}`
			`impure(a, d)`
				`def impure(x, y)`
				`  x[2] = 5`
				`  y["f"] = -1`
				`end`
				
				`a = [1, 2, 3]`
				`d = {"t" => 1, "f" => 0 }`
				`impure(a, d)`

	`return` *arg or* `undefined`. *If invoked with*			

return value	new *and return value not an object, returns* this	return *arg or* NULL	return *arg or* None	return *arg or last expression evaluated*

multiple return values	*none*	function first_and_second(&$a)
		{
		  return [$a[0], $a[1]];
		}
		
		$a = [1, 2, 3];
		list($x, $y) =
		  first_and_second($a);
			def first_and_second(a):
			  return a[0], a[1]
			
			x, y = first_and_second([1,2,3])
				def first_and_second(a)
				  return a[0], a[1]
				end
				
				x, y = first_and_second([1,2,3])

lambda declaration	var sqr = function(x) { return x*x; }	$sqr = function ($x) {
		  return $x * $x;
		};
			# body must be an expression:
			sqr = lambda x: x * x
				sqr = lambda { |x| x * x }

lambda invocation	sqr(2)	$sqr(2)	sqr(2)	sqr.call(2) *or* sqr[2]
function as value	var func = add;	$func = "add";	func = add	func = lambda {|*args| add(*args)}

function with private state	function counter() {
	  counter.i += 1;
	  return counter.i;
	}
	
	counter.i = 0;
	alert(counter());
		function counter()
		{
		  static $i = 0;
		  return ++$i;
		}
		
		echo counter();
			# state not private:
			def counter():
			  counter.i += 1
			  return counter.i
			
			counter.i = 0
			print(counter())
				none

closure	function make_counter() {
	  var i = 0;
	
	  return function() {
	    i += 1;
	    return i;
	  }
	}
		function make_counter()
		{
		  $i = 0;
		  return function () use (&$i) {
		    return ++$i;
		  };
		}
		
		$nays = make_counter();
		echo $nays();
			# Python 3:
			def make_counter():
			  i = 0
			  def counter():
			    nonlocal i
			    i += 1
			    return i
			  return counter
			
			nays = make_counter()
				def make_counter
				  i = 0
				  return lambda { i +=1; i }
				end
				
				nays = make_counter
				puts nays.call

generator		# PHP 5.5:
		function make_counter() {
		  $i = 0;
		  while (1) {
		    yield ++$i;
		  }
		}
		
		$nays = make_counter();
		# does not return a value:
		$nays->next();
		# runs generator if generator has not
		# yet yielded:
		echo $nays->current();
			# The itertools library contains
			# standard generators.
			# c.f. itertools.count()
			
			def make_counter():
			  i = 0
			  while True:
			    i += 1
			    yield i
			
			nays = make_counter()
			print(nays.next())
				def make_counter
				  return Fiber.new do
				    i = 0
				    while true
				      i += 1
				      Fiber.yield i
				    end
				  end
				end
				
				nays = make_counter
				puts nays.resume

decorator			def logcall(f):
			  def wrapper(*a, **opts):
			    print('calling ' + f.__name__)
			    f(*a, **opts)
			    print('called ' + f.__name__)
			  return wrapper
			
			@logcall
			def square(x):
			  return x * x	

			import operator	3.*(7)

operator as function			`operator.mul(3, 7)`
			
			`a = ['foo', 'bar', 'baz']`
			`operator.itemgetter(2)(a)`
				`a = ['foo', 'bar', 'baz']`
				`a.[](2)`

execution control

	javascript	php	python	ruby

if	`if (0 == n) {`
	`  alert("no hits");`
	`} else if (1 == n) {`
	`  alert("1 hit");`
	`} else {`
	`  alert(n + " hits");`
	`}`
		`if ( 0 == $n ) {`
		`  echo "no hits\n";`
		`} elseif ( 1 == $n ) {`
		`  echo "one hit\n";`
		`} else {`
		`  echo "$n hits\n";`
		`}`
			`if 0 == n:`
			`  print('no hits')`
			`elif 1 == n:`
			`  print('one hit')`
			`else:`
			`  print(str(n) + ' hits')`
				`if n == 0`
				`  puts "no hits"`
				`elsif 1 == n`
				`  puts "one hit"`
				`else`
				`  puts "#{n} hits"`
				`end`

switch	`switch (n) {`
	`case 0:`
	`  alert("no hits\n");`
	`  break;`
	`case 1:`
	`  alert("one hit\n");`
	`  break;`
	`default:`
	`  alert(3 + " hits\n");`
	`}`
		`switch ($n) {`
		`case 0:`
		`  echo "no hits\n";`
		`  break;`
		`case 1:`
		`  echo "one hit\n";`
		`  break;`
		`default:`
		`  echo "$n hits\n";`
		`}`
			none	`case n`
				`when 0`
				`  puts "no hits"`
				`when 1`
				`  puts "one hit"`
				`else`
				`  puts "#{n} hits"`
				`end`

while	`while (i < 100) {`
	`  i += 1;`
	`}`
		`while ( $i < 100 ) { $i++; }`	`while i < 100:`
			`  i += 1`
				`while i < 100 do`
				`  i += 1`
				`end`

c-style for	`for (var i = 0; i < 10; i++) {`
	`  alert(i);`
	`}`
		`for ($i = 1; $i <= 10; $i++) {`
		`  echo "$i\n";`
		`}`
			none	*none*

break, continue, redo	`break continue` *none*	`break continue` *none*	`break continue` *none*	`break next redo`
control structure keywords	`case catch debugger default do else finally for if switch throw try while`	`case default do else elseif for foreach goto if switch while`	`elif else for if while`	`case do else elsif end for loop when while unless until`
what do does	*starts body of a* do-while *loop, a loop which checks the condition after the body is executed*	*starts body of a* do-while *loop, a loop which checks the condition after the body is executed*	*raises* NameError *unless a value was assigned to it*	*starts an anonymous block. Also starts the body of a* loop, while, *or* until *loop*

statement modifiers	*none*	*none*	*none*	`puts "positive" if i > 0`
				`puts "nonzero" unless i == 0`

exceptions

	javascript	php	python	ruby
base exception	*Any value can be thrown.*	`Exception`	`BaseException` *User-defined exceptions should subclass* `Exception`. *In Python 2 old-style classes can be thrown.*	`Exception` *User-defined exceptions should subclass* `StandardError`.

		`Exception`	`BaseException`
			`  SystemExit`
			`  KeyboardInterrupt`
			`  GeneratorExit`
			`  Exception`
			`    StopIteration`
			`    StandardError`
			`      BufferError`
			`      ArithmeticError`
			`        FloatingPointError`
				`Exception`
				`  NoMemoryError`
				`  ScriptError`
				`    LoadError`
				`    NotImplementedError`
				`    SyntaxError`
				`  SignalException`
				`  StandardError`

predefined exceptions	`Error` `EvalError` `RangeError` `ReferenceError` `SyntaxError` `TypeError` `URIError`	`LogicException` `BadFunctionCallException` `BadMethodCallException` `DomainException` `InvalidArgumentException` `LengthException` `OutOfRangeException` `RuntimeException` `OutOfBoundsException` `OverflowException` `RangeException` `UnderflowException` `UnexpectedValueException`	`OverflowError` `ZeroDivisionError` `AssertionError` `AttributeError` `EnvironmentError` `EOFError` `ImportError` `LookupError` `IndexError` `KeyError` `MemoryError` `NameError` `ReferenceError` `RuntimeError` `NotImplementedError` `SyntaxError` `SystemError` `TypeError` `ValueError` `UnicodeError` *Python 3 has a different tree*	`ArgumentError` `IOError` `EOFError` `IndexError` `LocalJumpError` `NameError` `RangeError` `RegexpError` `RuntimeError` `SecurityError` `SocketError` `SystemCallError` `Errno::*` `SystemStackError` `ThreadError` `TypeError` `ZeroDivisionError` `SystemExit` `fatal`
raise exception	`throw new Error("bad arg");`	`throw new Exception("bad arg");`	`raise Exception('bad arg')`	`# raises RuntimeError` `raise "bad arg"`
catch exception	`try {` `risky();` `}` `catch (e) {` `alert("risky failed: " + e.message);` `}`	`try {` `risky();` `} catch (Exception $e) {` `echo "risky failed: ",` `$e->getMessage(), "\n";` `}`	`try:` `risky()` `except:` `print('risky failed')`	`# catches StandardError` `begin` `risky` `rescue` `print "risky failed: "` `puts $!.message` `end`
re-raise exception	`try {` `throw new Error("bam!");` `}` `catch (e) {` `alert("re-raising...");` `throw e;` `}`		`try:` `raise Exception('bam!')` `except:` `print('re-raising...')` `raise`	`begin` `raise "bam!"` `rescue` `puts "re-raising…"` `raise` `end`
global variable for last exception	*none*	*none*	*last exception:* `sys.exc_info()[1]`	*last exception:* `$!` *backtrace array of exc.:* `$@` *exit status of child:* `$?`
define exception	`function Bam(msg) {` `this.message = msg;` `}` `Bam.prototype = new Error;`	`class Bam extends Exception` `{` `function __construct()` `{` `parent::__construct("bam!");` `}` `}`	`class Bam(Exception):` `def __init__(self):` `super(Bam, self).__init__('bam!')`	`class Bam < Exception` `def initialize` `super("bam!")` `end` `end`
catch exception by type	`try {` `throw new Bam("bam!");` `}` `catch (e) {` `if (e instanceof Bam) {` `alert(e.message);` `}` `else {` `throw e;` `}` `}`	`try {` `throw new Bam;` `} catch (Bam $e) {` `echo $e->getMessage(), "\n";` `}`	`try:` `raise Bam()` `except Bam as e:` `print(e)`	`begin` `raise Bam.new` `rescue Bam => e` `puts e.message` `end`
	`acquire_resource();` `try {` `risky();`	*PHP 5.5:* `acquire_resource();` `try {`	`acquire_resource()` `try:`	`acquire_resource` `begin`

finally/ensure	`}`
	`finally {`
	`  release_resource();`
	`}`
		`  risky();`
		`}`
		`finally {`
		`  release_resource();`
		`}`
			`  risky()`
			`finally:`
			`  release_resource()`
				`  risky`
				`ensure`
				`  release_resource`
				`end`

concurrency				
	javascript	**php**	**python**	**ruby**

start thread		*none*	`class sleep10(threading.Thread):`
			`  def run(self):`
			`    time.sleep(10)`
			
			`thr = sleep10()`
			`thr.start()`
				`thr = Thread.new { sleep 10 }`

wait on thread			`thr.join()`	`thr.join`

sheet two: file handles | files | directories | processes and environment | option parsing | libraries and namespaces | objects | polymorphism | reflection | net and web | unit tests | debugging and profiling | deployment

General

versions used

The versions used for testing code in the reference sheet.

show version

How to get the version.

php:

The function `phpversion()` will return the version number as a string.

python:

The following function will return the version number as a string:

```
import platform

platform.python_version()
```

ruby:

Also available in the global constant `RUBY_VERSION`.

implicit prologue

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

javascript:

`underscore.js` adds some convenience functions as attributes of an object which is normally stored in the underscore _ variable. E.g.:

```
_.map([1, 2, 3], function(n){ return n * n; });
```

cdnjs hosts underscore.js and other JavaScript libraries for situations where it is inconvenient to have the webserver host the libraries.

When using `underscore.js` with the Node REPL, there is a conflict, since the Node REPL uses the underscore `_` variable to store the result of the last evaluation.

```
$ npm install underscore

$ node

> var us = require('underscore'); _

> us.keys({"one": 1, "two": 2});
[ 'one', 'two' ]
```

python:

We assume that `os`, `re`, and `sys` are always imported.

Grammar and Invocation

interpreter

The customary name of the interpreter and how to invoke it.

php:

`php -f` will only execute portions of the source file within a <?php *php code* ?> tag as php code. Portions of the source file outside of such tags is not treated as executable code and is echoed to standard out.

If short tags are enabled, then php code can also be placed inside <? *php code* ?> and <?= *php code* ?> tags.

<?= *php code* ?> is identical to <?php echo *php code* ?>.

repl

The customary name of the repl.

php:

The `php -a` REPL does not save or display the result of an expression.

python:

The python repl saves the result of the last statement in _.

ruby:

`irb` saves the result of the last statement in _.

command line program

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

block delimiters

How blocks are delimited.

python:

Python blocks begin with a line that ends in a colon. The block ends with the first line that is not indented further than the initial line. Python raises an IndentationError if the statements in the block that are not in a nested block are not all indented the same. Using tabs in Python source code is unrecommended and many editors replace them automatically with spaces. If the Python interpreter encounters a tab, it is treated as 8 spaces.

The python repl switches from a `>>>` prompt to a … prompt inside a block. A blank line terminates the block.

Colons are also used to separate keys from values in dictionary literals and in sequence slice notation.

ruby:

Curly brackets {} delimit blocks. A matched curly bracket pair can be replaced by the `do` and `end` keywords. By convention curly brackets are used for one line blocks.

The `end` keyword also terminates blocks started by `def`, `class`, or `module`.

Curly brackets are also used for hash literals, and the #{ } notation is used to interpolate expressions into strings.

statement separator

How the parser determines the end of a statement.

php:

Inside braces statements must be terminated by a semicolon. The following causes a parse error:

```
<? if (true) { echo "true" } ?>
```

The last statement inside `<?= ?>` or `<? ?>` tags does not need to be semicolon terminated, however. The following code is legal:

```
<?= $a = 1 ?>
<? echo $a ?>
```

python:

Newline does not terminate a statement when:

- inside parens
- inside list [] or dictionary {} literals

Python single quote '' and double quote "" strings cannot contain newlines except as the two character escaped form \n. Putting a newline in these strings results in a syntax error. There is however a multi-line string literal which starts and ends with three single quotes ''' or three double quotes: """.

A newline that would normally terminate a statement can be escaped with a backslash.

ruby:

Newline does not terminate a statement when:

- inside single quotes '', double quotes "", backticks ``, or parens ()
- after an operator such as + or , that expects another argument

Ruby permits newlines in array [] or hash literals, but only after a comma , or associator =>. Putting a newline before the comma or associator results in a syntax error.

A newline that would normally terminate a statement can be escaped with a backslash.

source code encoding

How to identify the character encoding for a source code file.

Setting the source code encoding makes it possible to safely use non-ASCII characters in string literals and regular expression literals.

end-of-line comment

How to create a comment that ends at the next newline.

multiple line comment

How to comment out multiple lines.

python:

The triple single quote ''' and triple double quote """ syntax is a syntax for string literals.

Variables and Expressions

local variable

How to declare variables which are local to the scope defining region which immediately contain them.

php:

Variables do not need to be declared and there is no syntax for declaring a local variable. If a variable with no previous reference is accessed, its value is *NULL*.

python:

A variable is created by assignment if one does not already exist. If the variable is inside a function or method, then its scope is the body of the function or method. Otherwise it is a global.

ruby:

Variables are created by assignment. If the variable does not have a dollar sign ($) or ampersand (@) as its first character then its scope is scope defining region which most immediately contains it.

A lower case name can refer to a local variable or method. If both are defined, the local variable takes precedence. To invoke the method make the receiver explicit: e.g. self.*name*. However, outside of class and modules local variables hide functions because functions are private methods in the class *Object*. Assignment to *name* will create a local variable if one with that name does not exist, even if there is a method *name*.

regions which define lexical scope

A list of regions which define a lexical scope for the local variables they contain.

Local variables defined inside the region are only in scope while code within the region is executing. If the language does not have closures, then code outside the region has no access to local variables defined inside the region. If the language does have closures, then code inside the region can make local variables accessible to code outside the region by returning a reference.

A region which is *top level* hides local variables in the scope which contains it from the code it contains. A region can also be top level if the syntax requirements of the language prohibit it from being placed inside another scope defining region.

A region is *nestable* if it can be placed inside another scope defining region, and if code in the inner region can access local variables in the outer region.

php:

Only function bodies and method bodies define scope. Function definitions can be nested, but when this is done lexical variables in the outer function are not visible to code in the body of the inner function.

Braces can be used to set off blocks of codes in a manner similar to the anonymous blocks of Perl. However, these braces do not define a scope. Local variables created inside the braces will be visible to subsequent code outside of the braces.

Local variables cannot be created in class bodies.

python:

Only functions and methods define scope. Function definitions can be nested. When this is done, inner scopes have read access to variables defined in outer scopes. Attempting to write (i.e. assign) to a variable defined in an outer scope will instead result in a variable getting created in the inner scope. Python trivia question: what would happen if the following code were executed?

```
def foo():
    v = 1
    def bar():
        print(v)
        v = 2
        print(v)
    bar()

foo()
```

ruby:

The keywords *if*, *unless*, *case*, *while*, and *until* each define a block which is terminated by an *end* keyword, but none of these blocks have their own scope.

Anonymous functions can be created with the *lambda* keyword. Ruby anonymous blocks can be provided after a function invocation and are bounded by curly brackets { } or the *do* and *end* keywords. Both anonymous functions and anonymous blocks can have parameters which are specified at the start of the block within pipes. Here are some examples:

```
id = lambda { |x| x }

[3, 1, 2, 4].sort { |a,b| a <=> b }

10.times do |i|
  print "#{i}..."
end
```

The scope of the parameters of an anonymous block or anonymous function is local to the block or function body.

It is possible to mark variables as local, even when they are already defined in the containing scope. Such variables are listed inside the parameter pipes, separated from the parameters by a semicolon:

```
x = 3
noop = lambda { |; x| x = 15 }
noop.call
# x is still 3
```

global variable

How to declare and access a variable with global scope.

php:

A variable is global if it is used at the top level (i.e. outside any function definition) or if it is declared inside a function with the *global* keyword. A function must use the *global* keyword to access the global variable.

python:

A variable is global if it is defined at the top level of a file (i.e. outside any function definition). Although the variable is global, it must be imported individually or be prefixed with the module name prefix to be accessed from another file. To be accessed from inside a function or method it must be declared with the *global* keyword.

ruby:

A variable is global if it starts with a dollar sign: $.

constant

How to declare a constant.

php:

A constant can be declared inside a class:

```
class Math {
  const pi = 3.14;
}
```

Refer to a class constant like this:

```
Math::pi
```

ruby:

Capitalized variables contain constants and class/module names. By convention, constants are all caps and class/module names are camel case. The ruby interpreter does not prevent modification of constants, it only gives a warning. Capitalized variables are globally visible, but a full or relative namespace name must be used to reach them: e.g. Math::PI.

assignment

How to assign a value to a variable.

python:

If the variable on the left has not previously been defined in the current scope, then it is created. This may hide a variable in a containing scope.

Assignment does not return a value and cannot be used in an expression. Thus, assignment cannot be used in a conditional test, removing the possibility of using assignment (=) when an equality test (==) was intended. Assignments can nevertheless be chained to assign a value to multiple variables:

```
a = b = 3
```

ruby:

Assignment operators have right precedence and evaluate to the right argument, so they can be chained. If the variable on the left does not exist, then it is created.

parallel assignment

How to assign values to variables in parallel.

python:

The r-value can be a list or tuple:

```
nums = [1,2,3]
a,b,c = nums
more_nums = (6,7,8)
d,e,f = more_nums
```

Nested sequences of expression can be assigned to a nested sequences of l-values, provided the nesting matches. This assignment will set a to 1, b to 2, and c to 3:

```
(a,[b,c]) = [1,(2,3)]
```

This assignment will raise a `TypeError`:

```
(a,(b,c)) = ((1,2),3)
```

In Python 3 the splat operator `*` can be used to collect the remaining right side elements in a list:

```
x, y, *z = 1, 2        # assigns [] to z
x, y, *z = 1, 2, 3     # assigns [3] to z
x, y, *z = 1, 2, 3, 4  # assigns [3, 4] to z
```

ruby:

The r-value can be an array:

```
nums = [1,2,3]
a,b,c = nums
```

swap

How to swap the values held by two variables.

compound assignment

Compound assignment operators mutate a variable, setting it to the value of an operation which takes the previous value of the variable as an argument.

If `<OP>` is a binary operator and the language has the compound assignment operator `<OP>=`, then the following are equivalent:

```
x <OP>= y
x = x <OP> y
```

The compound assignment operators are displayed in this order:

First row: arithmetic operator assignment: addition, subtraction, multiplication, (float) division, integer division, modulus, and exponentiation.
Second row: string concatenation assignment and string replication assignment
Third row: logical operator assignment: and, or, xor
Fourth row: bit operator assignment: left shift, right shift, and, or, xor.

python:

Python compound assignment operators do not return a value and hence cannot be used in expressions.

increment and decrement

The C-style increment and decrement operators can be used to increment or decrement values. They return values and thus can be used in expressions. The prefix versions return the value in the variable after mutation, and the postfix version return the value before mutation.

Incrementing a value two or more times in an expression makes the order of evaluation significant:

```
x = 1;
foo(++x, ++x); // foo(2, 3) or foo(3, 2)?

x = 1;
y = ++x/++x;  // y = 2/3 or y = 3/2?
```

Python avoids the problem by not having an in-expression increment or decrement.

Ruby mostly avoids the problem by providing a non-mutating increment and decrement. However, here is a Ruby expression which is dependent on order of evaluation:

```
x = 1
y = (x += 1)/(x += 1)
```

php:

The increment and decrement operators also work on strings. There are postfix versions of these operators which evaluate to the value before mutation:

```
$x = 1;
$x++;
$x--;
```

ruby:

The Integer class defines `succ`, `pred`, and `next`, which is a synonym for `succ`.

The String class defines `succ`, `succ!`, `next`, and `next!`. `succ!` and `next!` mutate the string.

null

The null literal.

null test

How to test if a variable contains null.

php:

$v == NULL does not imply that *$v* is *NULL*, since any comparison between *NULL* and a falsehood will return true. In particular, the following comparisons are true:

```
$v = NULL;
if ($v == NULL) { echo "true"; }

$v = 0;
if ($v == NULL) { echo "sadly true"; }

$v = '';
if ($v == NULL) { echo "sadly true"; }
```

undefined variable access

The result of attempting to access an undefined variable.

php:

PHP does not provide the programmer with a mechanism to distinguish an undefined variable from a variable which has been set to NULL.

A test showing that `isset` is the logical negation of `is_null`.

python:

How to test if a variable is defined:

```
not_defined = False
try: v
except NameError:
  not_defined = True
```

ruby:

How to test if a variable is defined:

```
! defined?(v)
```

conditional expression

How to write a conditional expression. A ternary operator is an operator which takes three arguments. Since

condition ? *true value* : *false value*

is the only ternary operator in C, it is unambiguous to refer to it as *the* ternary operator.

python:

The Python conditional expression comes from Algol.

ruby:

The Ruby `if` statement is also an expression:

```
x = if x > 0
  x
else
  -x
end
```

Arithmetic and Logic

true and false

Literals for the booleans.

These are the return values of the relational operators.

php:

Any identifier which matches TRUE case-insensitive can be used for the TRUE boolean. Similarly for FALSE.

In general, PHP variable names are case-sensitive, but function names are case-insensitive.

When converted to a string for display purposes, TRUE renders as "1" and FALSE as "". The equality tests `TRUE == 1` and `FALSE == ""` evaluate as TRUE but the equality tests `TRUE === 1` and `FALSE === ""` evaluate as FALSE.

falsehoods

Values which behave like the false boolean in a conditional context.

Examples of conditional contexts are the conditional clause of an `if` statement and the test of a `while` loop.

python:

Whether a object evaluates to True or False in a boolean context can be customized by implementing a __nonzero__ (Python 2) or __bool__ (Python 3) instance method for the class.

logical operators

Logical and, or, and not.

php, ruby:

&& and || have higher precedence than assignment, compound assignment, and the ternary operator (?:), which have higher precedence than *and* and *or*.

relational operators

Equality, inequality, greater than, less than, greater than or equal, less than or equal.

php:

Most of the relational operators will convert a string to a number if the other operand is a number. Thus 0 == "0" is true. The operators === and !== do not perform this conversion, so 0 === "0" is false.

python:

Relational operators can be chained. The following expressions evaluate to true:

```
1 < 2 < 3
1 == 1 != 2
```

In general if A_i are expressions and op_i are relational operators, then

$A_1\ op_1\ A_2\ op_2\ A_3\ \ldots\ A_n\ op_n\ A_{n+1}$

is true if and only if each of the following is true

$A_1\ op_1\ A_2$
$A_2\ op_2\ A_3$
...
$A_n\ op_n\ A_{n+1}$

min and max

How to get the min and max.

three value comparison

Binary comparison operators which return -1, 0, or 1 depending upon whether the left argument is less than, equal to, or greater than the right argument.

The `<=>` symbol is called the spaceship operator.

arithmetic operators

The operators for addition, subtraction, multiplication, float division, integer division, modulus, and exponentiation.

integer division

How to get the integer quotient of two integers.

divmod

How to get the quotient and remainder with single function call.

integer division by zero

What happens when an integer is divided by zero.

float division

How to perform floating point division, even if the operands might be integers.

float division by zero

What happens when a float is divided by zero.

power

How to get the value of a number raised to a power.

sqrt

The square root function.

sqrt -1

The result of taking the square root of negative one.

transcendental functions

Some mathematical functions. Trigonometric functions are in radians unless otherwise noted. Logarithms are natural unless otherwise noted.

python:

Python also has *math.log10*. To compute the log of *x* for base *b*, use:

```
math.log(x)/math.log(b)
```

ruby:

Ruby also has *Math.log2*, *Math.log10*. To compute the log of *x* for base *b*, use

```
Math.log(x)/Math.log(b)
```

transcendental constants

Constants for π and Euler's constant.

float truncation

How to truncate a float to the nearest integer towards zero; how to round a float to the nearest integer; how to find the nearest integer above a float; how to find the nearest integer below a float; how to take the absolute value.

absolute value

How to get the absolute value of a number.

integer overflow

What happens when the largest representable integer is exceeded.

float overflow

What happens when the largest representable float is exceeded.

rational numbers

How to create rational numbers and get the numerator and denominator.

ruby:

Require the library *mathn* and integer division will yield rationals instead of truncated integers.

complex numbers

python:

Most of the functions in *math* have analogues in *cmath* which will work correctly on complex numbers.

random integer, uniform float, normal float

How to generate a random integer between 0 and 99, include, float between zero and one in a uniform distribution, or a float in a normal distribution with mean zero and standard deviation one.

set random seed, get and restore seed

How to set the random seed; how to get the current random seed and later restore it.

All the languages in the sheet set the seed automatically to a value that is difficult to predict. The Ruby MRI interpreter uses the current time and process ID, for example. As a result there is usually no need to set the seed.

Setting the seed to a hardcoded value yields a random but repeatable sequence of numbers. This can be used to ensure that unit tests which cover code using random numbers doesn't intermittently fail.

The seed is global state. If multiple functions are generating random numbers then saving and restoring the seed may be necessary to produce a repeatable sequence.

bit operators

The bit operators for left shift, right shift, and, inclusive or, exclusive or, and negation.

binary, octal, and hex literals

Binary, octal, and hex integer literals

radix

How to convert integers to strings of digits of a given base. How to convert such strings into integers.

python

Python has the functions `bin`, `oct`, and `hex` which take an integer and return a string encoding the integer in base 2, 8, and 16.

```
bin(42)
```

```
oct(42)
hex(42)
```

Strings

string type

The type or types using for strings.

php:

PHP assumes all strings have single byte characters.

python:

In Python 2.7 the `str` type assumes single byte characters. A separate `unicode` type is available for working with Unicode strings.

In Python 3 the `str` type supports multibtye characters and the `unicode` type has been removed.

There is a mutable `bytearray` type and an immutable `bytes` type for working with sequences of bytes.

ruby:

The {String}} type supports multibtye characters. All strings have an explicit `Encoding`.

string literal

The syntax for string literals.

ruby:

How to specify custom delimiters for single and double quoted strings. These can be used to avoid backslash escaping. If the left delimiter is (, [, or { the right delimiter must be),], or }, respectively.

```
s1 = %q(lorem ipsum)
s2 = %Q(#{s1} dolor sit amet)
```

newline in literal

Whether newlines are permitted in string literals.

python:

Newlines are not permitted in single quote and double quote string literals. A string can continue onto the following line if the last character on the line is a backslash. In this case, neither the backslash nor the newline are taken to be part of the string.

Triple quote literals, which are string literals terminated by three single quotes or three double quotes, can contain newlines:

```
'''This is
two lines'''

"""This is also
two lines"""
```

literal escapes

Backslash escape sequences for inserting special characters into string literals.

unrecognized backslash escape sequence		
	double quote	single quote
JavaScript		
PHP	preserve backslash	preserve backslash
Python	preserve backslash	preserve backslash
Ruby	drop backslash	preserve backslash

python:

When string literals have an `r` or `R` prefix there are no backslash escape sequences and any backslashes thus appear in the created string. The delimiter can be inserted into a string if it is preceded by a backslash, but the backslash is also inserted. It is thus not possible to create a string with an `r` or `R` prefix that ends in a backslash. The `r` and `R` prefixes can be used with single or double quotes:

```
r'C:\Documents and Settings\Admin'
r"C:\Windows\System32"
```

The \u*hhhh* escapes are also available inside Python 2 Unicode literals. Unicode literals have a *u* prefiix:

```
u'lambda: \u03bb'
```

This syntax is also available in Python 3.3, but not Python 3.2. In Python 3.3 it creates a string of type `str` which has the same features as the `unicode` type of Python 2.7.

here document

Here documents are strings terminated by a custom identifier. They perform variable substitution and honor the same backslash escapes as double quoted strings.

python:

Python lacks variable interpolation in strings. Triple quotes honor the same backslash escape sequences as regular quotes, so triple quotes can otherwise be used like here documents:

```
s = '''here document
there computer
'''
```

ruby:

Put the customer identifier in single quotes to prevent variable interpolation and backslash escape interpretation:

```
s = <<'EOF'
Ruby code uses #{var} type syntax
to interpolate variables into strings.
EOF
```

variable interpolation

How to interpolate variables into strings.

python:

`str.format` will take named or positional parameters. When used with named parameters `str.format` can mimic the variable interpolation feature of the other languages.

A selection of variables in scope can be passed explicitly:

```
count = 3
item = 'ball'
print('{count} {item}s'.format(
  count=count,
  item=item))
```

Python 3 has `format_map` which accepts a `dict` as an argument:

```
count = 3
item = 'ball'
print('{count} {item}s'.format_map(locals()))
```

expression interpolation

How to interpolate the result of evaluating an expression into a string.

format

How to create a string using a printf style format.

python:

The % operator will interpolate arguments into printf-style format strings.

The `str.format` with positional parameters provides an alternative format using curly braces {0}, {1}, … for replacement fields.

The curly braces are escaped by doubling:

```
'to insert parameter {0} into a format, use {{{0}}}'.format(3)
```

If the replacement fields appear in sequential order and aren't repeated, the numbers can be omitted:

```
'lorem {} {} {}'.format('ipsum', 13, 3.7)
```

are strings mutable?

Are strings mutable?

copy string

How to copy a string such that changes to the original do not modify the copy.

concatenate

The string concatenation operator.

replicate

The string replication operator.

translate case

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

capitalize

How to capitalize a string and the words in a string.

ruby:

Rails monkey patches the `String` class with the `titleize` method for capitalizing the words in a string.

trim

How to remove whitespace from the ends of a string.

pad

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

number to string

How to convert numeric data to string data.

string to number

How to convert string data to numeric data.

php:

PHP converts a scalar to the desired type automatically and does not raise an error if the string contains non-numeric data. If the start of the string is not numeric, the string evaluates to zero in a numeric context.

python:

float and int raise an error if called on a string and any part of the string is not numeric.

ruby:

to_i and to_f always succeed on a string, returning the numeric value of the digits at the start of the string, or zero if there are no initial digits.

join

How to concatenate the elements of an array into a string with a separator.

split

How to split a string containing a separator into an array of substrings.

See also scan.

python:

`str.split()` takes simple strings as delimiters; use `re.split()` to split on a regular expression:

```
re.split('\s+', 'do re mi fa')
re.split('\s+', 'do re mi fa', 1)
```

split in two

How to split a string in two.

javascript:

A regular expression is probably the best method for splitting a string in two:

```
var m = /^([^ ]+) (.+)/.exec("do re mi");
var first = m[1];
var rest = m[2];
```

This technique works when the delimiter is a fixed string:

```
var a = "do re mi".split(" ");
var first = a[0];
var rest = a.splice(1).join(" ");
```

python:

Methods for splitting a string into three parts using the first or last occurrence of a substring:

```
'do re mi'.partition(' ')         # returns ('do', ' ', 're mi')
'do re mi'.rpartition(' ')        # returns ('do re', ' ', 'mi')
```

split and keep delimiters

How to split a string with the delimiters preserved as separate elements.

length

How to get the length in characters of a string.

index of substring

How to find the index of the leftmost occurrence of a substring in a string; how to find the index of the rightmost occurrence.

extract substring

How to extract a substring from a string by index.

extract character

How to extract a character from a string by its index.

chr and ord

Converting characters to ASCII codes and back.

The languages in this reference sheet do not have character literals, so characters are represented by strings of length one.

to array of characters

How to split a string into an array of single character strings.

translate characters

How to apply a character mapping to a string.

delete characters

How to remove all specified characters from a string; how to remove all but the specified characters from a string.

squeeze characters

How to replace multiple adjacent occurrences of a character with a single occurrence.

Regular Expressions

- PHP PCRE Regexes
- Python re library: 2.7, 3.1
- Ruby Regexp

Regular expressions or regexes are a way of specifying sets of strings. If a string belongs to the set, the string and regex "match". Regexes can also be used to parse strings.

The modern notation for regexes was introduced by Unix command line tools in the 1970s. POSIX standardized the notation into two types: extended regexes and the more archaic basic regexes. Perl regexes are extended regexes augmented by new character class abbreviations and a few other features introduced by the Perl interpreter in the 1990s. All the languages in this sheet use Perl regexes.

Any string that doesn't contain regex metacharacters is a regex which matches itself. The regex metacharacters are: `[ ] . | ( ) * + ? { } ^ $ \`

character classes: [ ] .

A character class is a set of characters in brackets: `[ ]`. When used in a regex it matches any character it contains.

Character classes have their own set of metacharacters: `^ - \ ]`

The `^` is only special when it is the first character in the character class. Such a character class matches its complement; that is, any character not inside the brackets. When not the first character the `^` refers to itself.

The hyphen is used to specify character ranges: e.g. `0-9` or `A-Z`. When the hyphen is first or last inside the brackets it matches itself.

The backslash can be used to escape the above characters or the terminal character class delimiter: `]`. It can be used in character class abbreviations or string backslash escapes.

The period `.` is a character class abbreviation which matches any character except for newline. In all languages the period can be made to match all characters. PHP uses the `m` modifier. Python uses the `re.M` flag. Ruby uses the `s` modifier.

character class abbreviations:

abbrev	name	character class
\d	digit	[0-9]
\D	nondigit	[^0-9]
\h	*PHP:* horizontal whitespace character *Ruby:* hex digit	*PHP:* [\t] *Ruby:* [0-9a-fA-F]
\H	*PHP:* not a horizontal whitespace character *Ruby:* not a hex digit	*PHP:* [^ \t] *Ruby:* [^0-9a-fA-F]
\s	whitespace character	[\t\r\n\f]
\S	non whitespace character	[^ \t\r\n\f]
\v	vertical whitespace character	[\r\n\f]
\V	not a vertical whitespace character	[^\r\n\f]
\w	word character	[A-Za-z0-9_]
\W	non word character	[^A-Za-z0-9_]

alternation and grouping: | ( )

The vertical pipe | is used for alternation and parens () for grouping.

A vertical pipe takes as its arguments everything up to the next vertical pipe, enclosing paren, or end of string.

Parentheses control the scope of alternation and the quantifiers described below. They are also used for capturing groups, which are the substrings which matched parenthesized parts of the regular expression. Each language numbers the groups and provides a mechanism for extracting them when a match is made. A parenthesized subexpression can be removed from the groups with this syntax: `(?:`*expr*`)`

quantifiers: * + ? { }

As an argument quantifiers take the preceding regular character, character class, or group. The argument can itself be quantified, so that `^a{4}*$` matches strings with the letter a in multiples of 4.

quantifier	# of occurrences of argument matched
*	zero or more, greedy
+	one or more, greedy
?	zero or one, greedy
{m,n}	*m* to *n*, greedy
{n}	exactly *n*
{m,}	*m* or more, greedy
{,n}	zero to *n*, greedy
*?	zero or more, lazy
+?	one or more, lazy
{m,n}?	*m* to *n*, lazy
{m,}?	*m* or more, lazy
{,n}?	zero to *n*, lazy

When there is a choice, greedy quantifiers will match the maximum possible number of occurrences of the argument. Lazy quantifiers match the minimum possible number.

anchors: ^ $

anchor	matches
^	beginning of a string. In Ruby or when *m* modifier is used also matches right side of a newline
$	end of a string. In Ruby or when *m* modifier is used also matches left side of a newline
\A	beginning of the string
\b	word boundary. In between a \w and a \W character or in between a \w character and the edge of the string
\B	not a word boundary. In between two \w characters or two \W characters
\z	end of the string
\Z	end of the string unless it is a newline, in which case it matches the left side of the terminal newline

**escaping: **

To match a metacharacter, put a backslash in front of it. To match a backslash use two backslashes.

php:

PHP 5.3 still supports the EREG engine, though the functions which use it are deprecated. These include the `split` function and functions which start with `ereg`. The preferred functions are `preg_split` and the other functions with a `preg` prefix.

literal, custom delimited literal

The literal for a regular expression; the literal for a regular expression with a custom delimiter.

javascript:

The constructor for a regular expression is:

```
var rx = RegExp("lorem|ipsum");
```

php:

PHP regex literals are strings. The first character is the delimiter and it must also be the last character. If the start delimiter is (, {, or [the end delimiter must be), }, or], respectively.

Here are the signatures from the PHP manual for the preg functions used in this sheet:

```
array preg_split ( string $pattern , string $subject [, int $limit = -1 [, int $flags = 0 ]] )

int preg_match ( string $pattern , string $subject [, array &$matches [, int $flags = 0 [, int $offset

mixed preg_replace ( mixed $pattern , mixed $replacement , mixed $subject [, int $limit = -1 [, int &$

int preg_match_all ( string $pattern , string $subject [, array &$matches [, int $flags = PREG_PATTERN
```

python:

Python does not have a regex literal, but the `re.compile` function can be used to create regex objects.

Compiling regexes can always be avoided:

```
re.compile('\d{4}').search('1999')
re.search('\d{4}', '1999')

re.compile('foo').sub('bar', 'foo bar')
re.sub('foo', 'bar', 'foo bar')

re.compile('\w+').findall('do re me')
re.findall('\w+', 'do re me')
```

character class abbreviations

The supported character class abbreviations.

Note that `\h` refers to horizontal whitespace (i.e. a space or tab) in PHP and a hex digit in Ruby. Similarly `\H` refers to something that isn't horizontal whitespace in PHP and isn't a hex digit in Ruby.

anchors

The supported anchors.

match test

How to test whether a string matches a regular expression.

python:

The `re.match` function returns true only if the regular expression matches the beginning of the string. `re.search` returns true if the regular expression matches any substring of the of string.

ruby:

`match` is a method of both `Regexp` and `String` so can match with both

```
/1999/.match("1999")
```

and

```
"1999".match(/1999/)
```

When variables are involved it is safer to invoke the `Regexp` method because string variables are more likely to contain `nil`.

case insensitive match test

How to perform a case insensitive match test.

modifiers

Modifiers that can be used to adjust the behavior of a regular expression.

The lists are not comprehensive. For all languages except Ruby there are additional modifiers.

modifier	behavior
e	*PHP:* when used with preg_replace, the replacement string, after backreferences are substituted, is eval'ed as PHP code and the result is used as the replacement.
g	*JavaScript:* read all non-overlapping matches into an array.
i, re.I	*all:* ignores case. Upper case letters match lower case letters and vice versa.
m, re.M	*JavaScript, PHP, Python:* makes the ^ and $ match the right and left edge of newlines in addition to the beginning and end of the string. *Ruby:* makes the period . match newline characters.
o	*Ruby:* performs variable interpolation #{ } only once per execution of the program.
s, re.S	*PHP, Python:* makes the period . match newline characters.
x, re.X	*all:* ignores whitespace in the regex which permits it to be used for formatting.

Python modifiers are bit flags. To use more than one flag at the same time, join them with bit or: |

substitution

How to replace all occurrences of a matching pattern in a string with the provided substitution string.

php:

The number of occurrences replaced can be controlled with a 4th argument to `preg_replace`:

```
$s = "foo bar bar";
preg_replace('/bar/', "baz", $s, 1);
```

If no 4th argument is provided, all occurrences are replaced.

python:

The 3rd argument to `sub` controls the number of occurrences which are replaced.

```
s = 'foo bar bar'
re.compile('bar').sub('baz', s, 1)
```

If there is no 3rd argument, all occurrences are replaced.

ruby:

The *gsub* operator returns a copy of the string with the substitution made, if any. The *gsub!* performs the substitution on the original string and returns the modified string.

The *sub* and *sub!* operators only replace the first occurrence of the match pattern.

match, prematch, postmatch

How to get the substring that matched the regular expression, as well as the part of the string before and after the matching substring.

ruby:

The special variables `$&`, `` $` ``, and `$'` also contain the match, prematch, and postmatch.

group capture

How to get the substrings which matched the parenthesized parts of a regular expression.

ruby:

Ruby has syntax for extracting a group from a match in a single expression. The following evaluates to "1999":

```
"1999-07-08"[/(\d{4})-(\d{2})-(\d{2})/, 1]
```

named group capture

How to get the substrings which matched the parenthesized parts of a regular expression and put them into a dictionary.

For reference, we call the `(?P<foo>...)` notation *Python-style* and the `(?<foo>...)` notation *Perl-style*.

php:

PHP originally supported Python-style named groups since that was the style that was added to the PCRE regex engine. Perl-style named groups were added to PHP 5.2.

python:

The Python interpreter was the first to support named groups.

scan

How to return all non-overlapping substrings which match a regular expression as an array.

backreference in match and substitution

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

recursive regex

Examples of recursive regexes.

The examples match substrings containing balanced parens.

Date and Time

In ISO 8601 terminology, a *date* specifies a day in the Gregorian calendar and a *time* does not contain date information; it merely specifies a time of day. A data type which combines both date and time information is probably more useful than one which contains just date information or just time information; it is unfortunate that ISO 8601 doesn't provide a name for this entity. The word *timestamp* often gets used to denote a combined date and time. PHP and Python use the compound noun *datetime* for combined date and time values.

An useful property of ISO 8601 dates, times, and date/time combinations is that they are correctly ordered by a lexical sort on their string representations. This is because they are big-endian (the year is the leftmost element) and they used fixed-length fields for each term in the string representation.

The C standard library provides two methods for representing dates. The first is the UNIX epoch, which is the seconds since January 1, 1970 in UTC. If such a time were stored in a 32-bit signed integer, the rollover would happen on January 18, 2038.

The other method of representing dates is the `tm` struct, a definition of which can be found on Unix systems in `/usr/include/time.h`:

```
struct tm {
        int     tm_sec;         /* seconds after the minute [0-60] */
        int     tm_min;         /* minutes after the hour [0-59] */
        int     tm_hour;        /* hours since midnight [0-23] */
        int     tm_mday;        /* day of the month [1-31] */
        int     tm_mon;         /* months since January [0-11] */
        int     tm_year;        /* years since 1900 */
        int     tm_wday;        /* days since Sunday [0-6] */
        int     tm_yday;        /* days since January 1 [0-365] */
        int     tm_isdst;       /* Daylight Savings Time flag */
        long    tm_gmtoff;      /* offset from CUT in seconds */
        char    *tm_zone;       /* timezone abbreviation */
};
```

Python uses and exposes the `tm` struct of the standard library. Python has a module called `time` which is a thin wrapper to the standard library functions which operate on this struct. Here is how get a `tm` struct in Python:

```
import time

utc = time.gmtime(time.time())
t = time.localtime(time.time())
```

The `tm` struct is a low level entity, and interacting with it directly should be avoided. In the case of Python it is usually sufficient to use the `datetime` module instead.

date/time type

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

current date/time

How to get the combined date and time for the present moment in both local time and UTC.

to unix epoch, from unix epoch

How to convert the native date/time type to the Unix epoch which is the number of seconds since the start of January 1, 1970 UTC.

python:

The Python datetime object created by `now()` and `utcnow()` has no timezone information associated with it. The `strftime()` method assumes a receiver with no timezone information represents a local time. Thus it is an error to call `strftime()` on the return value of `utcnow()`.

Here are two different ways to get the current Unix epoch. The second way is faster:

```
import calendar
import datetime

int(datetime.datetime.now().strftime('%s'))
calendar.timegm(datetime.datetime.utcnow().utctimetuple())
```

Replacing `now()` with `utcnow()` in the first way, or `utcnow()` with `now()` in the second way produces an incorrect value.

current unix epoch

How to get the current time as a Unix epoch timestamp.

strftime

How to format a date/time as a string using the format notation of the *strftime* function from the standard C library. This same format notation is used by the Unix *date* command.

php:

PHP supports strftime but it also has its own time formatting system used by `date`, `DateTime::format`, and `DateTime::createFromFormat`. The letters used in the PHP time formatting system are described here.

default format example

Examples of how a date/time object appears when treated as a string such as when it is printed to standard out.

The formats are in all likelihood locale dependent. The provided examples come from a machine running Mac OS X in the Pacific time zone of the USA.

php:

It is a fatal error to treat a DateTime object as a string.

strptime

How to parse a date/time using the format notation of the `strptime` function from the standard C library.

parse date w/o format

How to parse a date without providing a format string.

result date subtraction

The data type that results when subtraction is performed on two combined date and time values.

add time duration

How to add a time duration to a date/time.

A time duration can easily be added to a date/time value when the value is a Unix epoch value.

ISO 8601 distinguishes between a time interval, which is defined by two date/time endpoints, and a duration, which is the length of a time interval and can be defined by a unit of time such as '10 minutes'. A time interval can also be defined by date and time representing the start of the interval and a duration.

ISO 8601 defines notation for durations. This notation starts with a 'P' and uses a 'T' to separate the day and larger units from the hour and smaller units. Observing the location relative to the 'T' is important for interpreting the letter 'M', which is used for both months and minutes.

local timezone

Do date/time values include timezone information. When a date/time value for the local time is created, how the local timezone is determined.

A date/time value can represent a local time but not have any timezone information associated with it.

On Unix systems processes determine the local timezone by inspecting the file `/etc/localtime`.

php:

The default timezone can also be set in the `php.ini` file.

```
date.timezone = "America/Los_Angeles"
```

Here is the list of timezones supported by PHP.

arbitrary timezone

How to convert a timestamp to the equivalent timestamp in an arbitrary timezone.

timezone name, offset from UTC, is daylight savings?

How to get time zone information: the name of the timezone, the offset in hours from UTC, and whether the timezone is currently in daylight savings.

Timezones are often identified by three or four letter abbreviations. As can be seen from the list, many of the abbreviations do not uniquely identify a timezone. Furthermore many of the timezones have been altered in the past. The Olson database (aka Tz database) decomposes the world into zones in which the local clocks have all been set to the same time since 1970 and it gives these zones unique names.

ruby:

The `Time` class has a `zone` method which returns the time zone abbreviation for the object. There is a `tzinfo` gem which can be used to create timezone objects using the Olson database name. This can in turn be used to convert between UTC times and local times which are daylight saving aware.

microseconds

How to get the microseconds component of a combined date and time value. The SI abbreviations for milliseconds and microseconds are `ms` and `μs`, respectively. The C standard library uses the letter `u` as an abbreviation for `micro`. Here is a struct defined in `/usr/include/sys/time.h`:

```
struct timeval {
  time_t       tv_sec;   /* seconds since Jan. 1, 1970 */
  suseconds_t  tv_usec;  /* and microseconds */
};
```

sleep

How to put the process to sleep for a specified number of seconds. In Python and Ruby the default version of `sleep` supports a fractional number of seconds.

php:

PHP provides `usleep` which takes an argument in microseconds:

```
usleep(500000);
```

timeout

How to cause a process to timeout if it takes too long.

Techniques relying on SIGALRM only work on Unix systems.

Arrays

What the languages call their basic container types:

	javascript	php	python	ruby
array		array	list, tuple, sequence	Array, Enumerable
dictionary		array	dict, mapping	Hash

javascript:

php:

PHP uses the same data structure for arrays and dictionaries.

python:

Python has the mutable *list* and the immutable *tuple*. Both are *sequences*. To be a *sequence*, a class must implement __getitem__, __setitem__, __delitem__, __len__, __contains__, __iter__, __add__, __mul__, __radd__, and __rmul__.

ruby:

Ruby provides an *Array* datatype. If a class defines an *each* iterator and a comparison operator <=>, then it can mix in the *Enumerable* module.

literal

Array literal syntax.

ruby:

The `%w` operator splits the following string on whitespace and creates an array of strings from the words. The character following the `%w` is the string delimiter. If the following character is (, [, or {, then the character which terminates the string must be),], or }.

The `%W` operator is like the `%w` operator, except that double-quote style `#{ }` expressions will be interpolated.

quote words

The quote words operator, which is a literal for arrays of strings where each string contains a single word.

size

How to get the number of elements in an array.

empty test

How to test whether an array is empty.

lookup

How to access a value in an array by index.

python:

A negative index refers to the *length* - *index* element.

```
>>> a = [1,2,3]
>>> a[-1]
3
```

ruby:

A negative index refers to to the *length* - *index* element.

update

How to update the value at an index.

out-of-bounds behavior

What happens when the value at an out-of-bounds index is referenced.

index of element

slice

How to slice a subarray from an array by specifying a start index and an end index; how to slice a subarray from an array by specifying an offset index and a length index.

python:

Slices can leave the first or last index unspecified, in which case the first or last index of the sequence is used:

```
>>> a=[1,2,3,4,5]
>>> a[:3]
[1, 2, 3]
```

Python has notation for taking every nth element:

```
>>> a=[1,2,3,4,5]
>>> a[::2]
[1, 3, 5]
```

The third argument in the colon-delimited slice argument can be negative, which reverses the order of the result:

```
>>> a = [1,2,3,4]
>>> a[::-1]
[4, 3, 2, 1]
```

slice to end

How to slice to the end of an array.

The examples take all but the first element of the array.

manipulate back

How to add and remove elements from the back or high index end of an array.

These operations can be used to use the array as a stack.

manipulate front

How to add and remove elements from the front or low index end of an array.

These operations can be used to use the array as a stack. They can be used with the operations that manipulate the back of the array to use the array as a queue.

concatenate

How to create an array by concatenating two arrays; how to modify an array by concatenating another array to the end of it.

replicate

How to create an array containing the same value replicated *n* times.

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.

python:

The slice operator can be used to make a shallow copy:

```
a2 = a[:]
```

`list(v)` always returns a list, but `v[:]` returns a value of the same as `v`. The slice operator can be used in this manner on strings and tuples but there is little incentive to do so since both are immutable.

`copy.copy` can be used to make a shallow copy on types that don't support the slice operator such as a dictionary. Like the slice operator `copy.copy` returns a value with the same type as the argument.

arrays as function arguments

How arrays are passed as arguments.

iterate over elements

How to iterate through the elements of an array.

iterate over indices and elements

How to iterate through the elements of an array while keeping track of the index of each element.

iterate over range

Iterate over a range without instantiating it as a list.

instantiate range as array

How to convert a range to an array.

Python 3 ranges and Ruby ranges implement some of the functionality of arrays without allocating space to hold all the elements.

python:

In Python 2 `range()` returns a list.

In Python 3 `range()` returns an object which implements the immutable sequence API.

ruby:

The Range class includes the Enumerable module.

reverse

How to create a reversed copy of an array, and how to reverse an array in place.

python:

`reversed` returns an iterator which can be used in a `for/in` construct:

```
print("counting down:")
for i in reversed([1,2,3]):
  print(i)
```

`reversed` can be used to create a reversed list:

```
a = list(reversed([1,2,3]))
```

sort

How to create a sorted copy of an array, and how to sort an array in place. Also, how to set the comparison function when sorting.

php:

`usort` sorts an array in place and accepts a comparison function as a 2nd argument:

```
function cmp($x, $y) {
  $lx = strtolower($x);
  $ly = strtolower($y);
  if ( $lx < $ly ) { return -1; }
  if ( $lx == $ly ) { return 0; }
  return 1;
}

$a = ["b", "A", "a", "B"];

usort($a, "cmp");
```

python:

In Python 2 it is possible to specify a binary comparision function when calling `sort`:

```
a = [(1, 3), (2, 2), (3, 1)]

a.sort(cmp=lambda a, b: -1 if a[1] < b[1] else 1)

# a now contains:
[(3, 1), (2, 2), (1, 3)]
```

In Python 3 the `cmp` parameter was removed. One can achieve the same effect by defining `cmp` method on the class of the list element.

dedupe

How to remove extra occurrences of elements from an array.

python:

Python sets support the `len`, `in`, and `for` operators. It may be more efficient to work with the result of the set constructor directly rather than convert it back to a list.

membership

How to test for membership in an array.

intersection

How to compute an intersection.

python:

Python has literal notation for sets:

```
{1,2,3}
```

Use `set` and `list` to convert lists to sets and vice versa:

```
a = list({1,2,3})
ensemble = set([1,2,3])
```

ruby:

The intersect operator `&` always produces an array with no duplicates.

union

ruby:

The union operator `|` always produces an array with no duplicates.

relative complement, symmetric difference

How to compute the relative complement of two arrays or sets; how to compute the symmetric difference.

ruby:

If an element is in the right argument, then it will not be in the return value even if it is contained in the left argument multiple times.

map

Create an array by applying a function to each element of a source array.

ruby:

The `map!` method applies the function to the elements of the array in place.

`collect` and `collect!` are synonyms for `map` and `map!`.

filter

Create an array containing the elements of a source array which match a predicate.

ruby:

The in place version is `select!`.

`reject` returns the complement of `select`. `reject!` is the in place version.

The `partition` method returns two arrays:

```
a = [1, 2, 3]
lt2, ge2 = a.partition { |n| n < 2 }
```

reduce

Return the result of applying a binary operator to all the elements of the array.

python:

`reduce` is not needed to sum a list of numbers:

```
sum([1,2,3])
```

ruby:

The code for the reduction step can be provided by name. The name can be a symbol or a string:

```
[1,2,3].inject(:+)

[1,2,3].inject("+")

[1,2,3].inject(0, :+)

[1,2,3].inject(0, "+")
```

universal and existential tests

How to test whether a condition holds for all members of an array; how to test whether a condition holds for at least one member of any array.

A universal test is always true for an empty array. An existential test is always false for an empty array.

A existential test can readily be implemented with a filter. A universal test can also be implemented with a filter, but it is more work: one must set the condition of the filter to the negation of the predicate and test whether the result is empty.

shuffle and sample

How to shuffle an array. How to extract a random sample from an array.

php:

The `array_rand` function returns a random sample of the indices of an array. The result can easily be converted to a random sample of array values:

```
$a = [1, 2, 3, 4];
$sample = [];
foreach (array_rand($a, 2) as $i) { array_push($sample, $a[$i]); }
```

flatten

How to flatten nested arrays by one level or completely.

When nested arrays are flattened by one level, the depth of each element which is not in the top level array is reduced by one.

Flattening nested arrays completely leaves no nested arrays. This is equivalent to extracting the leaf nodes of a tree.

php, python:

To flatten by one level use reduce. Remember to handle the case where an element is not array.

To flatten completely write a recursive function.

zip

How to interleave arrays. In the case of two arrays the result is an array of pairs or an associative list.

Dictionaries

literal

The syntax for a dictionary literal.

size

How to get the number of dictionary keys in a dictionary.

lookup

How to lookup a dictionary value using a dictionary key.

out-of-bounds behavior

What happens when a lookup is performed on a key that is not in a dictionary.

python:

Use `dict.get()` to avoid handling `KeyError` exceptions:

```
d = {}
d.get('lorem')      # returns None
d.get('lorem', '')  # returns ''
```

is key present

How to check for the presence of a key in a dictionary without raising an exception. Distinguishes from the case where the key is present but mapped to null or a value which evaluates to false.

delete entry

How to remove a key/value pair from a dictionary.

from array of pairs, from even length array

How to create a dictionary from an array of pairs; how to create a dictionary from an even length array.

merge

How to merge the values of two dictionaries.

In the examples, if the dictionaries `d1` and `d2` share keys then the values from `d2` will be used in the merged dictionary.

invert

How to turn a dictionary into its inverse. If a key 'foo' is mapped to value 'bar' by a dictionary, then its inverse will map the key 'bar' to the value 'foo'. However, if multiple keys are mapped to the same value in the original dictionary, then some of the keys will be discarded in the inverse.

iteration

How to iterate through the key/value pairs in a dictionary.

python:

In Python 2.7 `dict.items()` returns a list of pairs and `dict.iteritems()` returns an iterator on the list of pairs.

In Python 3 `dict.items()` returns an iterator and `dict.iteritems()` has been removed.

keys and values as arrays

How to convert the keys of a dictionary to an array; how to convert the values of a dictionary to an array.

python:

In Python 3 `dict.keys()` and `dict.values()` return read-only views into the dict. The following code illustrates the change in behavior:

```
d = {}
keys = d.keys()
d['foo'] = 'bar'

if 'foo' in keys:
  print('running Python 3')
else:
  print('running Python 2')
```

sort by values

How to iterate through the key-value pairs in the order of the values.

default value, computed value

How to create a dictionary with a default value for missing keys; how to compute and store the value on lookup.

php:

Extend `ArrayObject` to compute values on lookup:

```
class Factorial extends ArrayObject {

  public function offsetExists($i) {
    return true;
  }

  public function offsetGet($i) {
    if(!parent::offsetExists($i)) {
      if ( $i < 2 ) {
        parent::offsetSet($i, 1);
      }
      else {
        $n = $this->offsetGet($i-1);
        parent::offsetSet($i, $i*$n);
      }
    }
    return parent::offsetGet($i);
  }
}

$factorial = new Factorial();
```

Functions

Python has both functions and methods. Ruby only has methods: functions defined at the top level are in fact methods on a special main object. Perl subroutines can be invoked with a function syntax or a method syntax.

define function

How to define a function.

invoke function

How to invoke a function.

python:

Parens are mandatory, even for functions which take no arguments. Omitting the parens returns the function or method as an object. Whitespace can occur between the function name and the following left paren.

In Python 3 print is a function instead of a keyword; parens are mandatory around the argument.

ruby:

Ruby parens are optional. Leaving out the parens results in ambiguity when function invocations are nested. The interpreter resolves the ambiguity by assigning as many arguments as possible to the innermost function invocation, regardless of its actual arity. It is mandatory that the left paren not be separated from the method name by whitespace.

apply function to array

How to apply a function to an array.

perl:

Perl passes the elements of arrays as individual arguments. In the following invocation, the function `foo()` does not know which arguments came from which array. For that matter it does not know how many arrays were used in the invocation:

```
foo(@a, @b);
```

If the elements must be kept in their respective arrays the arrays must be passed by reference:

```
sub foo {
  my @a = @{$_[0]};
  my @b = @{$_[1]};
}

foo(\@a, \@b);
```

When hashes are used as arguments, each key and value becomes its own argument.

missing argument behavior

How incorrect number of arguments upon invocation are handled.

default argument

How to declare a default value for an argument.

variable number of arguments

How to write a function which accepts a variable number of argument.

python:

This function accepts one or more arguments. Invoking it without any arguments raises a `TypeError`:

```
def poker(dealer, *players):
  ...
```

ruby:

This function accepts one or more arguments. Invoking it without any arguments raises an `ArgumentError`:

```
def poker(dealer, *players)
  ...
end
```

named parameters

How to write a function which uses named parameters and how to invoke it.

python:

The caller can use named parameter syntax at the point of invocation even if the function was defined using positional parameters.

The splat operator * collects the remaining arguments into a list. In a function invocation, the splat can be used to expand an array into separate arguments.

The double splat operator ** collects named parameters into a dictionary. In a function invocation, the double splat expands a dictionary into named parameters.

A double splat operator can be used to force the caller to use named parameter syntax. This method has the disadvantage that spelling errors in the parameter name are not caught:

```
def fequal(x, y, **kwargs):
  eps = opts.get('eps') or 0.01
  return abs(x - y) < eps
```

In Python 3 named parameters can be made mandatory:

```
def fequal(x, y, *, eps):
  return abs(x-y) < eps

fequal(1.0, 1.001, eps=0.01)  # True

fequal(1.0, 1.001)                 # raises TypeError
```

pass number or string by reference

How to pass numbers or strings by reference.

The three common methods of parameter passing are *pass by value*, *pass by reference*, and *pass by address*. Pass by value is the default in most languages.

When a parameter is passed by reference, the callee can changed the value in the variable that was provided as a parameter, and the caller will see the new value when the callee returns. When the parameter is passed by value the callee cannot do this.

When a language has mutable data types it can be unclear whether the language is using pass by value or pass by reference.

pass array or dictionary by reference

How to pass an array or dictionary without making a copy of it.

return value

How the return value of a function is determined.

multiple return values

How to return multiple values from a function.

lambda declaration and invocation

How to define and invoke a lambda function.

python:

Python lambdas cannot contain newlines or semicolons, and thus are limited to a single statement or expression. Unlike named functions, the value of the last statement or expression is returned, and a *return* is not necessary or permitted. Lambdas are closures and can refer to local variables in scope, even if they are returned from that scope.

If a closure function is needed that contains more than one statement, use a nested function:

```
def make_nest(x):
    b = 37
    def nest(y):
        c = x*y
        c *= b
        return c
    return nest

n = make_nest(12*2)
print(n(23))
```

Python closures are read only.

A nested function can be returned and hence be invoked outside of its containing function, but it is not visible by its name outside of its containing function.

ruby:

The following lambda and Proc object behave identically:

```
sqr = lambda { |x| x * x }

sqr = Proc.new {|x| x * x }
```

With respect to control words, Proc objects behave like blocks and lambdas like functions. In particular, when the body of a Proc object contains a `return` or `break` statement, it acts like a `return` or `break` in the code which invoked the Proc object. A `return` in a lambda merely causes the lambda to exit, and a `break` inside a lambda must be inside an appropriate control structure contained with the lambda body.

Ruby are alternate syntax for defining lambdas and invoking them:

```
sqr = ->(x) {x*x}

sqr.(2)
```

function as value

How to store a function in a variable and pass it as an argument.

php:

If a variable containing a string is used like a function then PHP will look for a function with the name in the string and attempt to invoke it.

python:

Python function are stored in variables by default. As a result a function and a variable with the same name cannot share the same scope. This is also the reason parens are mandatory when invoking Python functions.

function with private state

How to create a function with private state which persists between function invocations.

python:

Here is a technique for creating private state which exploits the fact that the expression for a default value is evaluated only once:

```
def counter(_state=[0]):
  _state[0] += 1
  return _state[0]

print(counter())
```

closure

How to create a first class function with access to the local variables of the local scope in which it was created.

python:

Python 2 has limited closures: access to local variables in the containing scope is read only and the bodies of anonymous functions must consist of a single expression.

Python 3 permits write access to local variables outside the immediate scope when declared with `nonlocal`.

generator

How to create a function which can yield a value back to its caller and suspend execution.

python:

A Python generator is a function which returns an iterator.

An iterator is an object with two methods: `iter()`, which returns the iterator itself, and `next()`, which returns the next item or raises a `StopIteration` exception.

Python sequences, of which lists are an example, define an `iter()` for returned an iterator which traverses the sequence.

Python iterators can be used in *for/in* statements and list comprehensions.

In the table below, `p` and `q` are variables for iterators.

itertools	
generator	**description**
count(start=0, step=1)	`arithmetic sequence of integers`
cyle(p)	`cycle over p endlessly`
repeat(v, [n])	`return v n times, or endlessly`
chain(p, q)	`p followed by q`
compress(p, q)	`p if q`
groupby(p, func)	
ifilter(pred, p)	`p if pred(p)`
ifilterfalse(pred, p)	`p if not pred(p)`
islice(p, [start], stop, [step])	
imap	
starmap()	
tee()	
takewhile()	
izip()	
izip_longest()	
product()	
permutations()	
combinations()	
combinations_with_replacement()	

ruby:

Ruby generators are called fibers.

decorator

A decorator replaces an invocation of one function with another in a way that that is imperceptible to the client.

Normally a decorator will add a small amount of functionality to the original function which it invokes. A decorator can modify the arguments before passing them to the original function or modify the return value before returning it to the client. Or it can leave the arguments and return value unmodified but perform a side effect such as logging the call.

operator as function

How to call an operator using the function invocation syntax.

This can be useful when dealing with an API which accepts a function as an argument.

python:

The `operator` module provides functions which perform the same operations as the various operators. Using these functions is more efficient than wrapping the operators in lambdas.

ruby:

All operators can be invoked with method invocation syntax. The binary operator invocation syntax can be regarded as syntactic sugar.

Execution Control

if

The `if` statement.

php:

PHP has the following alternate syntax for `if` statements:

```
if ($n == 0):
  echo "no hits\n";
elseif ($n == 1):
  echo "one hit\n";
else:
  echo "$n hits\n";
endif;
```

ruby:

If an `if` statement is the last statement executed in a function, the return value is the value of the branch that executed.

Ruby `if` statements are expressions. They can be used on the right hand side of assignments:

```
m = if n
  1
else
  0
end
```

switch

The `switch` statement.

while

php:

PHP provides a `do-while` loop. The body of such a loop is guaranteed to execute at least once.

```
$i = 0;
do {
    echo $i;
} while ($i > 0);
```

ruby:

Ruby provides a loop with no exit condition:

```
def yes(expletive="y")
  loop do
   puts expletive
  end
end
```

Ruby also provides the `until` loop.

Ruby loops can be used in expression contexts but they always evaluate to `nil`.

c-style for

How to write a C-style for loop.

break, continue, redo

break exits a *for* or *while* loop immediately. *continue* goes to the next iteration of the loop. *redo* goes back to the beginning of the current iteration.

control structure keywords

A list of control structure keywords. The loop control keywords from the previous line are excluded.

The list summarizes the available control structures. It excludes the keywords for exception handling, loading libraries, and returning from functions.

what do does

How the `do` keyword is used.

statement modifiers

Clauses added to the end of a statement to control execution.

Ruby has conditional statement modifiers. Ruby also has looping statement modifiers.

ruby:

Ruby has the looping statement modifiers `while` and `until`:

```
i = 0
i += 1 while i < 10

j = 10
j -= 1 until j < 0
```

Exceptions

base exception

predefined exceptions

raise exception

How to raise exceptions.

ruby:

Ruby has a *throw* keyword in addition to *raise*. *throw* can have a symbol as an argument, and will not convert a string to a RuntimeError exception.

catch exception

How to catch exceptions.

php:

PHP code must specify a variable name for the caught exception. *Exception* is the top of the exception hierarchy and will catch all exceptions.

Internal PHP functions usually do not throw exceptions. They can be converted to exceptions with this signal handler:

```
function exception_error_handler($errno, $errstr, $errfile, $errline ) {
    throw new ErrorException($errstr, 0, $errno, $errfile, $errline);
}
set_error_handler("exception_error_handler");
```

ruby:

A *rescue Exception* clause will catch any exception. A *rescue* clause with no exception type specified will catch exceptions that are subclasses of *StandardError*. Exceptions outside *StandardError* are usually unrecoverable and hence not handled in code.

In a *rescue* clause, the *retry* keyword will cause the *begin* clause to be re-executed.

In addition to *begin* and *rescue*, ruby has *catch*:

```
catch (:done) do
  loop do
    retval = work
    throw :done if retval < 10
  end
end
```

re-raise exception

How to re-raise an exception preserving the original stack trace.

python:

If the exception is assigned to a variable in the `except` clause and the variable is used as the argument to `raise`, then a new stack trace is created.

ruby:

If the exception is assigned to a variable in the `rescue` clause and the variable is used as the argument to `raise`, then the original stack trace is preserved.

global variable for last exception

The global variable name for the last exception raised.

define exception

How to define a new variable class.

catch exception by type

How to catch exceptions of a specific type and assign the exception a name.

php:

PHP exceptions when caught must always be assigned a variable name.

finally/ensure

Clauses that are guaranteed to be executed even if an exception is thrown or caught.

Concurrency

start thread

ruby:

Ruby MRI threads are operating system threads, but a global interpreter lock prevents more than one thread from executing Ruby code at a time.

wait on thread

How to make a thread wait for another thread to finish.

JavaScript

Mozilla Developer Network: JavaScript

Browsers

Mozilla Developer Network: Document
Mozilla Document Network: Window
W3C: Document Object Model (DOM) Level 3 Core Specification
W3C: Document Object Model (DOM) Level 3 Events Specification

Most browsers include a debugger which can be launched with a keystroke:

browser	mac	windows	linux
Chrome	⌥⌘J	Cmd+Shift+J	Cmd+Shift+J
Firefox	⌥⌘S	Cmd+Shift+S	Cmd+Shift+S
Safari	⌥⌘C		

The debugger has a console pane, which is a JavaScript REPL.

```
> Math.log(10)
2.302585092994046

> alert("Hello, World!")
```

The console provides a global object named `document` which provides access to the DOM of the current page:

```
> document.getElementsByTagName("div").length
302
```

TODO: more ways to select node elements. Attributes of node elements.

There is also a global object named `window` which is useful.

JavaScript can be embedded in an HTML document using the `<script>` tag:

```
<script>
  var sum = 1 + 2;
  alert('the sum is ' + sum);
</script>
```

Alternatively the JavaScript can be in a separate file served by the same server:

```
<script src="foo.js"></script>
```

`<script>` tags can be placed in either the `<head>` or the `<body>` of an `<html>` document. They are executed as they are encountered by the browser. If there is a syntax error, then none of the JavaScript in the `<script>` tag is executed. If there is an unhandled exception, the browser stops execution of the `<script>` at that point. Neither syntax errors nor unhandled exceptions prevent the browser from executing JavaScript in subsequent `<script>` tags.

using JavaScript to modify the DOM

waiting for all JavaScript to load

javascript URL

DOM events

To guard against websites serving malicious JavaScript code, the JavaScript interpreters in browsers do not provide the ability to interact with the local operating system. In particular, client-side JavaScript cannot read or write to files. Client-side JavaScript cannot spawn other processes.

Client-side JavaScript can make HTTP requests. Client-side JavaScript can modify the DOM of an HTML page which was served from the same origin as the JavaScript. To be from the same origin, the URLs must have the same protocol, domain, and port. Client-side JavaScript can also get and set cookies which share the same origin. The origin policy for cookies is slightly relaxed, since the JavaScript can also get and set cookies for a parent domain, excluding public top level domains such as `.com`, `.net`, and `.org`.

Node

Node.js

using node

asynchronous programming

PHP

PHP Manual

The PHP interpreter is packaged in 3 different ways: (1) as a standalone executable which can be executed as a CGI script, (2) as a dynamically linked library which adheres to the SAPI of a webserver such as Apache or IIS, and (3) as a standalone executable which can be used to run PHP scripts from the command line. The latter executable is called PHP CLI.

From the perspective of a PHP programmer, there no important differences between PHP CGI and PHP SAPI. The programmer should be aware of the following differences between PHP CGI/SAPI and PHP CLI:

- PHP CGI/SAPI writes HTTP headers to standard out before any output specified by the program. PHP CLI does not.
- PHP CLI sets the constants STDIN, STDOUT, and STDERR. PHP CGI/SAPI do not.
- PHP CLI has no timeout. PHP CGI/SAPI will typically timeout a script after 30 seconds.
- PHP CGI/SAPI add HTML markup to error messages. PHP CLI does not.

- PHP CLI does not buffer output, so calling `flush` is never necessary. PHP CGI/SAPI buffer output.

Python

2.7: Language, Standard Library
3.2: Language, Standard Library

Python uses leading whitespace to indicate block structure. It is not recommended to mix tabs and spaces in leading whitespace, but when this is done, a tab is equal to 8 spaces. The command line options '-t' and '-tt' will warn and raise an error respectively when tabs are used inconsistently for indentation.

Regular expressions and functions for interacting with the operating system are not available by default and must be imported to be used, i.e.

```
import re, sys, os
```

Identifiers in imported modules must be fully qualified unless imported with *from/import*:

```
from sys import path
from re import *
```

There are two basic sequence types: the mutable list and the immutable tuple. The literal syntax for lists uses square brackets and commas [1,2,3] and the literal syntax for tuples uses parens and commas (1,2,3).

The dictionary data type literal syntax uses curly brackets, colons, and commas { “hello”:5, “goodbye”:7 }. Python 3 adds a literal syntax for sets which uses curly brackets and commas: {1,2,3}. This notation is also available in Python 2.7. Dictionaries and sets are implemented using hash tables and as a result dictionary keys and set elements must be hashable.

All values that can be stored in a variable and passed to functions as arguments are objects in the sense that they have methods which can be invoked using the method syntax.

Attributes are settable by default. This can be changed by defining a __setattr__ method for the class. The attributes of an object are stored in the __dict__ attribute. Methods must declare the receiver as the first argument.

Classes, methods, functions, and modules are objects. If the body of a class, method, or function definition starts with is a string, it is available available at runtime via __doc__. Code examples in the string which are preceded with '>>>' (the python repl prompt) can be executed by doctest and compared with the output that follows.

Ruby

1.9.3 core, stdlib

Ruby has a type of value called a symbol. The literal syntax for a symbol is :*identifier* or :"*arbitrary string*". The methods `to_s` and `to_sym` can be used to convert symbols to strings and strings to symbols. Symbols can be used to pass functions or methods as arguments by name. They can be used as keys in Hash objects in place of strings, but the client must remember the type of the keys since `:foo != "foo"`. Also note that converting a Hash object with symbols as keys to JSON and then back will yield a Hash object with strings as keys.

In Ruby all values that can be stored in a variable and passed to functions as arguments are objects in the sense that they have methods which can be invoked using the method syntax. Moreover classes are objects. The system provided classes are open and as a result the user can add methods to classes such as `String`, `Array`, or `Fixnum`. Ruby only permits single inheritance, but Ruby modules are mix-ins and can be used to add methods to a class via the `include` statement.

Ruby methods can be declared private and this is enforced by the interpreter. Object attributes are private by default and attribute names have an ampersand @ prefix. The methods `attr_reader`, `attr_writer`, and `attr_accessor` can be used in a class block to define a getter, setter, or both for an attribute.

When invoking a method the parens are optional. If there are two or more arguments they must still be separated by commas. If one of the arguments is an expression containing a method invocation with arguments, then the Ruby interpreter will assign as many arguments as possible to the innermost method invocation.

Inside a Ruby method, the `self` keyword refers to the receiver. It is not declared when defining the method. Ruby functions are implemented as methods on an object called *main* which has the special property that any methods defined on it become instance methods in the `Object` class which is a base class of most Ruby objects. This makes the method available everywhere. Methods defined at the top level are also added to the *main* object and the `Object` class. Functions which Ruby provides by default are instance methods defined the `Object` class or the `Kernel` module.

Ruby methods are not objects and cannot directly be stored in variables. It is worth emphasizing that the Python interpreter when encountering a method identifier with no parens returns the method as an object value, but the Ruby interpreter invokes the method. As mentioned earlier, methods can be passed by name using symbols. If a method receives a symbol representing a method as an argument, it can invoke the method with the syntax :*symbol*.to_proc.call(*args…*). Note that `to_proc` resolves the symbol to the method that is in scope where it is invoked.

Although passing a method or a function is a bit awkward, Ruby provides a convenient mechanism called *blocks* for simultaneously defining an anonymous function at the invocation of a method and providing it to the method as an argument. The block appears immediately after the closing paren of the method invocation and uses either the { |*args…*| *body* } or `do` |*args…*| *body* `end` syntax. The invoked method can in turn invoke the block with the `yield` keyword.

Ruby blocks are closures like lambda functions and can see local variables in the enclosing scope in which they were defined. The parameters of the block are local to the block. Semicolon syntax is available so that identifiers listed after the arguments could be made local to the block even if already defined in the containing scope.

The `lambda` keyword or the `Proc.new` constructor can be used to store an anonymous function in a variable. The function can be invoked with *variable*.call(). If such a function is passed to a method argument as the last argument and preceded with an ampersand, the function will be used as the block for the method. Conversely, if the last argument in a method definition is preceded with an ampersand, any block provided to the function will be bound to the argument name as an anonymous function.