Scripting Languages: PHP, Perl, Python, Ruby

A side-by-side reference sheet; there is also a reference sheet for Tcl, Lua, JavaScript, Io

arithmetic and logic | strings | containers | functions | execution control | environment and i/o | libraries and modules | objects | reflection and hooks | web | web framework | java interop | history | edit

	php (1995)	perl (1987)	python (1991)	ruby (1995)
versions used	*5.3.1*	*5.10.0; 5.12.1*	*2.6.1; 3.1.1*	*1.8.7; 1.9.1*
show version	$ php --version	$ perl --version	$ python -V	$ ruby --version
interpreter	$ php -f foo.php	$ perl foo.pl	$ python foo.py	$ ruby foo.rb
repl	$ php -a	$ perl -de 0	$ python	$ irb
check syntax	$ php -l foo.php	$ perl -c foo.pl	# precompile to bytecode: import py_compile py_compile.compile("foo.py")	$ ruby -c foo.rb
flags for stronger and strongest warnings	*none*	$ perl -w foo.pl $ perl -W foo.pl	$ python -t foo.py $ python -3t foo.py	$ ruby -w foo.pl $ ruby -W2 foo.pl
statement separator	;	;	; # or sometimes newline	; # or sometimes newline
block delimiters	{}	{}	*offside rule*	{} do end

assignment	$a = 1;	$a = 1;	# does not return a value:
			a = 1
				a = 1

parallel assignment	list($a, $b, $c) = array(1 ,2, 3);
	# 3 is ignored:
	list($a, $b) = array(1, 2, 3);
	# $c set to NULL:
	list($a, $b, $c) = array(1, 2);
		($a, $b, $c) = (1, 2, 3);
		# 3 is ignored:
		($a, $b) = (1, 2, 3);
		# $c set to undef:
		($a, $b, $c) = (1, 2);
			a, b, c = 1, 2, 3
			# raises ValueError:
			a, b = 1, 2, 3
			# raises ValueError:
			a, b, c = 1, 2
				a, b, c = 1, 2, 3
				# 3 is ignored:
				a, b = 1, 2, 3
				# c set to nil:
				a, b, c = 1, 2

swap	list($a, $b) = array($b, $a);	($a, $b) = ($b, $a);	a, b = b, a	a, b = b, a
compound			# do not return values:	

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

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

local variable declarations	# in function body:
	$a = NULL;
	$b = array();
	$c = array();
	$d = 1;
	list($e, $f) = array(2, 3);
		my $a;
		my (@b, %c);
		my $d = 1;
		my ($e, $f) = (2, 3);
			# in function body:
			a = None
			b, c = [], {}
			d = 1
			e, f = 2, 3
				a = nil
				b, c = [], {}
				d = 1
				e, f = 2, 3

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

global variable	list($g, $h) = array(7, 8);
	function swap_globals() {
	global $g, $h;
	list($g, $h) = array($h, $g);
	}
		our ($g, $h) = (7, 8);
		sub swap_globals() {
		($g, $h) = ($h, $g);
		}
			g, h = 7, 8
			def swap_globals():
			global g, h
			g, h = h, g
				$g, $h = 7, 8
				def swap_globals()
				$g, $h = $h, $g
				end

constant declaration	class Math {
	const pi = 3.14;
	}
	# how to reference constant:
	Math::pi
		use constant PI => 3.14;	# uppercase identifiers
			# constant by convention
			PI = 3.14
				# warning if capitalized
				# identifier is reassigned
				PI = 3.14

to-end-of-line comment	// comment # comment	# comment	# comment	# comment
multiline comment	/* comment line another line */	=for comment line another line =cut	'''comment line another line'''	=begin comment line another line =end
null	NULL # case insensitive	undef	None	nil
null test	is_null($v) ! isset($v)	! defined $v	v == None v is None	v == nil v.nil?

undefined variable access	NULL	*error under* use strict; *otherwise* undef	*raises* NameError	*raises* NameError
undefined test	*same as null test; no distinction between undefined variables and variables set to* NULL	*same as null test; no distinction between undefined variables and variables set to* undef	not_defined = False try: v except NameError: not_defined = True	! defined?(v)

arithmetic and logic

	php	perl	python	ruby
true and false	TRUE FALSE # case insensitve	1 0	True False	true false
falsehoods	FALSE NULL 0 0.0 '' '0' array()	undef 0 0.0 '' '0' ()	False None 0 0.0 '' [] {}	false nil
logical operators	&& || ! *lower precedence:* and or xor	and or not *also:* && || !	and or not	and or not *also:* && || !
conditional expression	$x > 0 ? $x : -$x	$x > 0 ? $x : -$x	x if x > 0 else -x	x > 0 ? x : -x
comparison operators	== != *or* <> > < >= <= *no conversion:* === !==	*numbers only:* == != > < >= <= *strings:* eq ne gt lt ge le	== != > < >= <=	== != > < >= <=
convert from string, to string	7 + '12' 73.9 + '.037' 'value: ' . 8	7 + '12' 73.9 + '.037' 'value: ' . 8	7 + int('12') 73.9 + float('.037') 'value: ' + str(8)	7 + "12".to_i 73.9 + ".037".to_f "value: " + "8".to_s
arithmetic operators	+ - * / *none* % pow(b,e)	+ - * / *none* % **	+ - * / // % **	+ - * x.fdiv(y) / % **
integer division	(int) ($a / $b)	int ($a / $b)	a // b	a / b
float division	$a / $b	$a / $b	float(a) / b # python 3: a / b	a.to_f / b *or* a.fdiv(b)
arithmetic functions	sqrt exp log sin cos tan asin acos atan atan2	sqrt exp log sin cos *none none none none* atan2	from math import sqrt, exp, log, \ sin, cos, tan, asin, acos, atan, atan2	include Math sqrt exp log sin cos tan asin acos atan atan2
arithmetic truncation	abs($x) round($x) ceil($x) floor($x)	abs($x) *none* *POSIX:* ceil($x) floor($x)	import math abs(x) int(round(x)) math.ceil(x) math.floor(x)	x.abs x.round x.ceil x.floor
	min(1,2,3) max(1,2,3)	use List::Util qw(min max); min(1,2,3);	min(1,2,3)	

min and max	# of an array:
	$a = array(1,2,3)
	call_user_func_array(min, $a)
	call_user_func_array(max, $a)
		max(1,2,3);
		@a = (1,2,3);
		min(@a);
		max(@a);
			max(1,2,3)
			min([1,2,3])
			max([1,2,3])
				[1,2,3].min
				[1,2,3].max

division by zero	*returns zero with warning*	*error*	*raises* ZeroDivisionError	*integer division raises* ZeroDivisionError *float division returns* Infinity
integer overflow	*converted to float*	*converted to float*	*becomes arbitrary length integer of type* long	*becomes arbitrary length integer of type* Bignum
float overflow	INF	inf	*raises* OverflowError	Infinity
sqrt -2	NaN	*error*	# raises ValueError: import math math.sqrt(-2) # returns complex float: import cmath cmath.sqrt(-2)	*raises* Errno::EDOM
rational numbers	*none*	*none*	from fractions import Fraction x = Fraction(22,7) x.numerator x.denominator	require 'rational' x = Rational(22,7) x.numerator x.denominator
complex numbers	*none*	*none*	z = 1 + 1.414j z.real z.imag	require 'complex' z = 1 + 1.414.im z.real z.imag
random integer, uniform float, normal float	rand(0,99) lcg_value() *none*	int(rand() * 100) rand() *none*	import random random.randint(0,99) random.random() random.gauss(0,1)	rand(100) rand *none*
bit operators	<< >> & | ^ ~	<< >> & | ^ ~	<< >> & | ^ ~	<< >> & | ^ ~
strings				
	php	**perl**	**python**	**ruby**
character literal	*none*	*none*	*none*	*none*
chr and ord	chr(65) ord("A")	chr(65) ord("A")	chr(65) ord('A')	65.chr "A".ord
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\""
newline in literal	*yes*	*yes*	*no, use escape or triple quote literal*	*yes*
	$computer = 'PC';	$computer = 'PC';		computer = 'PC'

here document	$s = <<<EOF here document there $computer EOF;	$s = <<EOF; here document there $computer EOF	*none*	s = <<EOF here document there #{computer} EOF
escapes	*single quoted:* \' \\ *double quoted:* \f \n \r \t \v \x*hh* \$ \" \o*oo*	*single quoted:* \' \\ *double quoted:* \a \b \c*x* \e \f \n \r \t \x*hh* \x{*hhhh*} \o*oo*	*newline* \\ \' \" \a \b \f \n \r \t \v \o*oo* \x*hh* # python 3: \u*hhhh*	# single quoted: \' \\ # double quoted: \a \b \c*x* \e \f \n \r \s \t \u*hhhh* \u{*hhhhh*} \v \x*hh* \o*oo*
encoding				
variable interpolation	$count = 3; $item = "ball"; echo "$count ${item}s\n";	my $count = 3; my $item = "ball"; print "$count ${item}s\n";	*none*	count = 3 item = "ball" puts "#{count} #{item}s"
length	strlen("hello")	length("hello")	len('hello')	"hello".length "hello".size
character count	$a = count_chars("(3*(7+12))"); $a[ord('(')]	"(3*(7+12))" =~ tr/(//	'(3*(7+12))'.count('(')	'(3*(7+12))'.count('(')
index of substring	strpos("foo bar", "bar")	index("foo bar","bar")	'foo bar'.index('bar')	"foo bar".index("bar")
extract substring	substr("foo bar", 4, 3)	substr("foo bar",4,3)	"foo bar"[4:7]	"foo bar"[4,3]
concatenate	"hello, " . "world"	"hello, " . "world"	'hello, ' + 'world'	"hello, " + "world"
replicate	$hbar = str_repeat('-', 80);	my $hbar = '-' x 80;	hbar = '-' * 80	hbar = '-' * 80
split	explode(" ","foo bar baz") preg_split('/\s+/',"foo bar baz")	split(/\s+/,"foo bar baz")	'foo bar baz'.split()	"foo bar baz".split
join	$a = array("foo","bar","baz"); implode(" ", $a)	join(' ',("foo","bar","baz"))	' '.join(['foo','bar','baz'])	['foo','bar','baz'].join(' ')
scan	$s = "foo bar baz"; preg_match_all('/\w+/', $s, $a); $a[0]	*none*	import re s = 'foo bar baz' re.compile('\w+').findall(s)	"foo bar baz".scan(/\w+/)
pack and unpack	$f="a3ilfd"; $s=pack($f,"hello",7,7,3.14,3.14); unpack("a3a/ii/ll/ff/dd",$s);	my ($f, $s); $f="a5ilfd"; $s=pack($f,"hello",7,7,3.14,3.14); unpack($f,$s)	import struct f='5silfd' s=struct.pack(f,'hello',7,7,3.14,3.14) struct.unpack(f,s)	f="a5ilfd" s=["hello",7,7,3.14,3.14].pack(f) s.unpack(f)

sprintf	$fmt = "foo: %s %d %f";
	sprintf($fmt, 'bar', 13, 3.7);
		my $fmt = "foo: %s %d %f";
		sprintf($fmt, 'bar', 13, 3.7)
			'foo: %s %d %f' % ('bar',13,3.7)
			# new in python 2.6:
			fmt = 'foo: {0} {1} {2}'
			str.format(fmt, 'bar', 13, 3.7)
				"foo: %s %d %f" % ['bar',13,3.7]

	strtoupper("hello")	uc("hello")	'hello'.upper()	"hello".upcase

case manipulation	strtolower("HELLO") ucfirst("hello")	lc("HELLO") ucfirst("hello")	'HELLO'.lower() 'hello'.capitalize()	"HELLO".downcase "hello".capitalize
strip	trim(" foo ") ltrim(" foo") rtrim("foo ")	*use regex substitution*	' foo '.strip() ' foo'.lstrip() 'foo '.rstrip()	" foo ".strip " foo".lstrip "foo ".rstrip
pad on right, on left	str_pad("hello", 10) str_pad("hello", 10, " ", STR_PAD_LEFT)	sprintf("%-10s","hello") sprintf("%10s","hello")	'hello'.ljust(10) 'hello'.rjust(10)	"hello".ljust(10) "hello".rjust(10)
character translation	$ins = implode(range('a','z')); $outs = substr($ins,13,13) . substr($ins,0,13); strtr("hello",$ins,$outs)	$s = "hello"; $s =~ tr/a-z/n-za-m/;	from string import lowercase as ins from string import maketrans outs = ins[13:] + ins[:13] "hello".translate(maketrans(ins,outs))	"hello".tr('a-z','n-za-m')
regexp match	preg_match('/^\d{4}$/',"1999") preg_match('/^[a-z]+/',"foo BAR") preg_match('/[A-Z]+/',"foo BAR")	"1999" =~ /^\d{4}$/ "foo BAR" =~ /^[a-z]+/ "foo BAR" =~ /[A-Z]+/	import re re.match("\d{4}$","1999") re.match("[a-z]+","foo BAR") re.search("[A-Z]+","foo BAR")	"1999".match(/^\d{4}$/) "foo BAR".match(/^[a-z]+/) "foo BAR".match(/[A-Z]+/)

match, prematch, postmatch	*none*	my $s = "A 17 B 12";
		while ($s =~ /\d+/) {
		my $discard = $`;
		my $number = $&;
		$s = $';
		print $number . "\n";
		}
			s = "A 17 B 12"
			while True:
			m = re.search('\d+',s)
			if not m:
			break
			discard = s[0:m.start(0)]
			number = m.group()
			s = s[m.end(0):len(s)]
			print(s)
				s = "A 17 B 12"
				while (/\d+/.match(s)) do
				discard = $`
				number = $&
				s = $'
				puts number
				end

substring matches	$a = array();
	$s = "2010-06-03";
	$r = '/(\d{4})-(\d{2})-(\d{2})/';
	preg_match($r, $s, $a);
	list($_, $yr, $mn, $dy) = $a;
		"2010-06-03" =~ /(\d{4})-(\d{2})-(\d{2})/;
		($yr, $mn, $dy) = ($1, $2, $3);
			import re
			reg = "(\d{4})-(\d{2})-(\d{2})"
			m = re.search(reg, "2010-06-03")
			yr,mn,dy = m.groups()
				reg = /(\d{4})-(\d{2})-(\d{2})/
				m = reg.match("2010-06-03")
				yr,mn,dy = m[1..3]

single substitution	$s = 'foo bar bar'; preg_replace('/bar/','baz',$s,1);	$s = "foo bar bar"; $s =~ s/bar/baz/; $s	import re s = 'foo bar bar' re.compile('bar').sub('baz', s, 1)	"foo bar bar".sub(/bar/,'baz')
global substitution	$s = 'foo bar bar'; preg_replace('/bar/', 'baz', $s);	$s = "foo bar bar"; $s =~ s/bar/baz/g; $s	import re s = 'foo bar bar' re.compile('bar').sub('baz', s)	"foo bar bar".gsub(/bar/,'baz')

containers

	php	perl	python	ruby
array literal	$nums = array(1,2,3,4);	@nums = (1,2,3,4);	nums = [1,2,3,4]	nums = [1,2,3,4]
array size	count($nums)	$#nums + 1 *or* scalar(@nums)	len(nums)	nums.size nums.length # same as size
array lookup	$nums[0]	$nums[0]	nums[0]	nums[0]

	$nums[0]	$nums[0]	nums[0]	nums[0]
index of array element	array_search(2, array(1,2,3))	*none*	[1,2,3].index(2)	[1,2,3].index(2)
array slice	# 3rd arg is length of slice: array_slice($nums,1,2)	@nums[1..2]	nums[1:3]	nums[1..2]

manipulate back of array	$a = array(6,7,8);
	array_push($a, 9);
	array_pop($a);
		@a = (6,7,8);
		push @a, 9;
		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 of array	$a = array(6,7,8);
	array_unshift($a, 5);
	array_shift($a);
		@a = (6,7,8);
		unshift @a, 5;
		shift @a;
			a = [6,7,8]
			a.insert(0,5)
			a.pop(0)
				a = [6,7,8]
				a.unshift(5)
				a.shift

array concatenation	$a = array(1,2,3); $b = array_merge($a,array(4,5,6)); $a = array_merge($a,array(4,5,6));	@a = (1,2,3); @b = (@a,(4,5,6)); push @a, (4,5,6);	a = [1,2,3] b = a + [4,5,6] a.extend([4,5,6])	a = [1,2,3] b = a + [4,5,6] a.concat([4,5,6])
address copy, shallow copy, deep copy	$a = (1,2,array(3,4)); $b =& $a; *none* $d = $a;	@a = (1,2,[3,4]); $b = \@a; @c = @a; *none*	a = [1,2,[3,4]] b = a c = list(a) import copy d = copy.deepcopy(a)	a = [1,2,[3,4]] b = a c = a.dup d = Marshal.load(Marshal.dump(a))
arrays as function arguments	*parameter contains deep copy*	*each element passed as separate argument; use reference to pass array as single argument*	*parameter contains address copy*	*parameter contains address copy*

array iteration	foreach (array(1,2,3) as $i) {
	  echo "$i\n";
	}
		for $i (1 2 3) { print "$i\n" }	for i in [1,2,3]:
			  print(i)
				[1,2,3].each { |i| puts i }

indexed array iteration	$a = array('a','b','c');
	foreach ($a as $i => $c) {
	  echo "$c at index $i\n";
	}
		@a = ('a','b','c');
		for ($i=0; $i<scalar(@a); $i++) {
		  print "$a[$i] at index $i\n";
		};
			for i, c in enumerate(['a','b','c']):
			  print("%s at index %d" % (c, i))
				a = ['a','b','c']
				a.each_with_index do |c,i|
				  puts "#{c} at index #{i}"
				end

sort	$a = array(3,1,4,2); *none* sort($a);	@a = (3,1,4,2); sort @a; @a = sort @a; sort { $a <=> $b } @a;	a = [3,1,4,2] sorted(a) a.sort()	a = [3,1,4,2] a.sort a.sort! a.sort { |m,n| m <=> n}
reverse	$a = array(1,2,3); array_reverse($a); $a = array_reverse($a);	@a = (1,2,3); reverse @a; @a = reverse @a;	a = [1,2,3] list(reversed([1,2,3])) a.reverse()	a = [1,2,3] a.reverse a.reverse!
membership	in_array(7, $nums)	grep { 7 == $_ } @nums	7 in nums	nums.include?(7)
intersection	$a = array(1,2); $b = array(2,3,4) array_intersect($a, $b)		set.intersection(set([1,2]), set([2,3,4]))	[1,2] & [2,3,4]

union			set.union(set([1,2]),set([2,3,4]))	[1,2] | [2,3,4]
map	$t2 = create_function('$x', 'return $x*$x;') array_map($t2, array(1,2,3))	map { $_ * $_ } (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	$gt1 = create_function('$x','return $x>1;'); array_filter(array(1,2,3), $gt1)	grep { $_ > 1 } (1,2,3)	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	$add = create_function('$a,$b','return $a+$b;'); array_reduce(array(1,2,3),$add,0)	reduce { $a + $b } 0, (1,2,3)	# import needed in python 3 only import reduce from functools reduce(lambda x,y:x+y,[1,2,3],0)	[1,2,3].inject(0) { |m,o| m+o }
universal test	*none, use array_filter*	*none, use grep*	all(i%2 == 0 for i in [1,2,3,4])	[1,2,3,4].all? {|i| i.even? }
existential test	*none, use array_filter*	*none, use grep*	any(i%2 == 0 for i in [1,2,3,4])	[1,2,3,4].any? {|i| i.even? }
dictionary literal	$h = array('t' => 1, 'f' => 0);	%h = ('t' => 1, 'f' => 0);	h = { 't':1, 'f':0 }	h = { 't' => 1, 'f' => 0 }
dictionary size	count($h)	scalar(keys %h)	len(h)	h.size h.length # same as size
dictionary lookup	$h['t']	$h{'t'}	h['t']	h['t']
is dictionary key present	array_key_exists('y',$h);	defined($h{'y'})	'y' in h	h.has_key?('y')
delete dictionary entry	$h = array(1 => 't', 0 => 'f'); unset($h[1]);	%h = (1 => 't', 0 => 'f'); delete $h{1};	h = {1: True, 0: False} del h[1]	h = {1 => true, 0 => false} h.delete(1)

dictionary iteration	foreach ($h as $k => $v) {	while (($k, $v) = each %h) {	# python 2:
			for k,v in h.iteritems():
			  code
			# python 3:
			for k,v in h.items():
			  code
				h.each { |k,v| *code* }

keys and values of dictionary as arrays	array_keys($h) array_values($h)	keys %h values %h	h.keys() h.values()	h.keys h.values
out of bounds behavior	NULL	undef	*raises* IndexError *or* KeyError	nil

functions

	php	perl	python	ruby
function declaration	function add($a, $b) { return $a + $b; }	sub add { $_[0] + $_[1] }	def add(a,b): return a+b	def add(a,b); a+b; end

function invocation	add(3,7);	add(1,2);	add(1,2)	add(1,2)
missing argument	*set to* NULL *with warning*	*set to* undef	*raises* TypeError	*raises* ArgumentError
default value	function my_log($x, $b=10) {	*none*	def log(x,base=10):	def log(x,base=10)

arbitrary number of arguments	function add() { return array_sum(func_get_args()); }	*@_ contains all values*	def add(first,*rest):
			  if not rest:
			  return first
			  else:
			  return first+add(*rest)
				def add(first, *rest)
				  if rest.empty?
				  first
				  else
				  first + add(*rest)
				  end
				end

named parameter definition	*none*	*none*	def f(**d):	def f(h)
named parameter invocation	*none*	*none*	f(eps=0.01)	f(:eps => 0.01)

pass number or string by reference	function foo(&$x, &$y) {
	  $x += 1;
	  $y .= 'ly';
	}
	
	$n = 7;
	$s = 'hard';
	foo($n, $s);
		sub foo {
		  $_[0] += 1;
		  $_[1] .= 'ly';
		}
		
		my $n = 7;
		my $s = 'hard';
		foo($n, $s);
			not possible	*not possible*

pass array or hash by reference	function foo(&$x, &$y) {
	  $x[2] = 5;
	  $y['f'] = -1;
	}
	
	$a = array(1,2,3);
	$h = array('t'=>1,'f'=>0);
	foo($a, $h);
		sub foo {
		  $_[0][2] = 5;
		  $_[1]{'f'} = -1;
		}
		
		my @a = (1,2,3);
		my %h = ('t'=> 1, 'f' => 0);
		foo(\@a, \%h);
			def foo(x, y):
			  x[2] = 5
			  y['f'] = -1
			
			a = [1,2,3]
			h = {'t':1, 'f':0}
			foo(a, h)
				def foo(x, y)
				  x[2] = 5
				  y['f'] = -1
				end
				
				a = [1,2,3]
				h = {'t'=> 1, 'f' => 0 }
				foo(a, h)

return value	return *arg or* NULL	return *arg or last expression evaluated*	return *arg or* None	return *arg or last expression evaluated*

multiple return values	function first_and_second(&$a) {
	  return array($a[0], $a[1]);
	}
	$a = array(1,2,3);
	list($x, $y) =
	  first_and_second($a);
		sub first_and_second {
		  return ($_[0], $_[1]);
		}
		@a = (1,2,3);
		($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	$f = create_function('$x','return	$f = sub { $_[0] * $_[0] }	f = lambda x: x * x	f = lambda { |x| x * x }

	$x*$x;');	$f = sub { $_[0] * $_[0] }	f = lambda x: x * x	f = lambda { |x| x * x }
lambda invocation	$f(2)	$f->(2)	f(2)	f.call(2)

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

closure	function make_counter() {
	  $i = 0;
	  return function () use (&$i) {
	  return ++$i;
	  };
	}
	$nays = make_counter();
	echo $nays();
		sub make_counter() {
		  my $i = 0;
		  return sub() { ++$i };
		}
		my $nays = make_counter();
		print $nays->() . "\n"
			# 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	*none*	*none*	def make_counter():
			  i = 0
			  while True:
			  i += 1
			  yield(i)
			nays = make_counter()
			print(nays.next())
				# ruby 1.9:
				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

execution control				
	php	**perl**	**python**	**ruby**

if	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\n"
		} elsif (1 == $n) {
		  print "1 hit\n"
		} else {
		  print "$n hits\n"
		}
			if 0 == n:
			  print("no hits")
			elif 1 == n:
			  print("1 hit")
			else:
			  print(str(n) + " hits")
				if n == 0
				  puts "no hits"
				elsif 1 == n
				  puts "1 hit"
				else
				  puts "#{n} hits"
				end

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

break/continue/redo	break continue *none*	last next redo	break continue *none*	break next redo

	break continue *none*	last next redo	break continue *none*	break next redo

for	for ($i = 1; $i <= 10; $i++) {
	  echo "$i\n";
	}
		for ($i=0; $i <= 10; $i++) {
		  print "$i\n";
		}
			none	*none*

range iteration	foreach (range(1,10) as $i) {
	  echo "$i\n";
	}
		for $i (1..10) { print "$i\n" }	for i in range(1,11):
			  print(i)
				(1..10).each { |i| puts i }

statement modifiers	*none*	print "positive\n" if $i > 0; print "nonzero\n" unless $i == 0;	*none*	puts 'positive' if i > 0 puts 'nonzero' unless i == 0
raise exception	throw new Exception("bad arg");	die "bad arg";	raise Exception("bad arg")	# raises RuntimeError raise "bad arg"

catch exception	try {
	  risky();
	} catch (Exception $e) {
	  echo 'risky failed: ',
	  $e->getMessage(), "\n";
	}
		eval { risky };
		if ($@) {
		  print "risky failed: $@\n";
		}
			try:
			  risky()
			except:
			  print("risky failed")
				# catches StandardError
				begin
				  risky
				rescue
				  print "risky failed: "
				  puts $!.message
				end

global variable for last exception raised	*none*	*$EVAL_ERROR: $@* *$OS_ERROR: $!* *$CHILD_ERROR: $?*	*none*	*last exception: $!* *backtrace array of exc.: $@* *exit status of child: $?*

define exception	class Bam extends Exception {
	  function __construct() {
	  parent::__construct("bam!");
	  }
	}
		none	class Bam(Exception):
			  def __init__(self):
			  super(Bam, self).__init__("bam!")
				class Bam < Exception
				  def initialize
				  super("bam!")
				  end
				end

catch exception by type and assign to variable	try {
	  throw new Bam;
	} catch (Bam $e) {
	  echo $e->getMessage(), "\n";
	}
		none	try:
			  raise Bam()
			except Bam as e:
			  print(e)
				begin
				  raise Bam.new
				rescue Bam => e
				  puts e.message
				end

finally/ensure	*none*	*none*	try:
			  acquire_resource()
			  risky()
			finally:
			  release_resource()
				begin
				  acquire_resource
				  risky
				ensure
				  release_resource
				end

uncaught exception behavior		*stderr and exit*	*stderr and exit*	*stderr and exit*

start thread	*none*	use threads;
		$f = sub { sleep 10 };
		$t = threads->new($f);
			class sleep10(threading.Thread):
			  def run(self):
			  time.sleep(10)
			t = sleep10()
				t = Thread.new { sleep 10 }

			t.start()	
wait on thread	*none*	$t->join;	t.join()	t.join

environment and i/o

	php	perl	python	ruby
external command	exec('ls');	system('ls');	import os os.system('ls')	system('ls')
backticks	exec('ls', $out = array()); $out	`ls`;	import os os.popen('ls').read()	`ls`
command line args, script name	count($argv) $argv[0] $argv[1] ... $_SERVER["SCRIPT_NAME"]	$#ARGV + 1 $ARGV[0] $ARGV[1] ... $0	import sys len(sys.argv)-1 sys.argv[1] sys.argv[2] ... sys.argv[0]	ARGV.size ARGV[0] ARGV[1] ... $0
print to standard out	echo "hi world\n";	print "hi world\n";	print("hi world")	puts "hi world"
standard file handles	$stdin = fopen('php://stdin','r'); $stdout = fopen('php://stdout','w'); $stderr = fopen('php://stderr','w');	STDIN STDOUT STDERR	import sys sys.stdin sys.stdout sys.stderr	$stdin $stdout $stderr
open file	$f = fopen('/etc/hosts','r');	open FILE, '/etc/hosts';	f = open('/etc/hosts')	f = File.open('/etc/hosts') *or* File.open('/etc/hosts') { |f|
open file for writing	$f = fopen('/tmp/php_test','w');	open FILE, ">/tmp/perl_test";	f = open('/tmp/test','w')	f = File.open('/tmp/test','w') *or* File.open('/tmp/test','w') { |f|
close file	fclose($f);	close FILE;	f.close()	f.close
read line	$line = fgets($f);	$line = <FILE>	f.readline()	f.gets
iterate over a file by line	while (!feof($f)) { $line = fgets($f);	while ($line = <FILE>) {	for line in f:	f.each do |line|
chomp	chop($line);	chomp $line;	line = line.rstrip('\r\n')	line.chomp!
read entire file into array or string	$a = file('/etc/hosts'); $s = file_get_contents('/etc/hosts');	@a = <FILE>; $/ = undef; $s = <FILE>;	a = f.readlines() s = f.read()	a = f.lines.to_a s = f.read
write to file	fwrite($f, 'hello');	print FILE "hello";	f.write('hello')	f.write('hello')
flush file		use IO::Handle; FILE->flush();	f.flush()	f.flush
environment variable	getenv('HOME')	$ENV{'HOME'}	import os	ENV['HOME']

variable	getenv('HOME')	$ENV{HOME}	os.getenv('HOME')	ENV['HOME']
exit	exit 0;	exit 0;	import sys sys.exit(0)	exit(0)

set signal handller		$SIG{INT} = sub {
		  die "exiting...\n";
		};
			import signal
			def handler(signo, frame):
			  print("exiting...")
			  exit -1
			signal.signal(signal.SIGINT, handler)
				Signal.trap("INT", lambda { |signo| puts "exiting..."; exit })

libraries and modules

	php	perl	python	ruby
load library	require_once('foo.php');	require 'Foo.pm'; # or require Foo; # or use Foo;	import foo	require 'foo' # or require 'foo.rb'
reload library	require('foo.php');	do 'Foo.pm';	reload(foo)	load 'foo.rb'
library path	$o = ini_get('include_path'); $n = $o . ':/some/path'; ini_set('include_path', $n);	push @INC, '/some/path';	import sys sys.path.append('/some/path')	$: << '/some/path'
library path environment variable	*none*	PERL5LIB	PYTHONPATH	RUBYLIB
library path command line option	*none*	-I	*none*	-I

module declaration	namespace Foo;	package Foo;
		require Exporter;
		our @ISA = ("Exporter");
		our @EXPORT_OK = ('bar', 'baz');
			put declarations in foo.py	class Foo *or* module Foo

submodule declaration	namespace Foo\Bar;	package Foo::Bar;	*create directory* foo *in library path containing file* bar.py	module Foo::Bar *or* module Foo module Bar
module separator	\Foo\Bar\baz();	Foo::Bar::baz();	foo.bar.baz()	Foo::Bar.baz
import module	*none, but a long module name can be shortened*	# imports symbols in @EXPORT: use Foo;	from foo import *	
import symbols	*only class names can be imported*	# bar and baz must be in # @EXPORT or @EXPORT_OK: use Foo ('bar', 'baz');	from foo import bar, baz	*none*

package			$ python
			>>> help('modules')
			download pkg with matching python	

package management	$ pear list
	$ pear install Math_BigInteger
		$ perldoc perllocal
		$ perl -MCPAN -e 'install Moose'
			download pkg with matching python version
			$ tar xf libxml2-python-2.6.0.tar.gz
			$ cd libxml2-python-2.6.0
			$ sudo python setup.py install
				$ gem list
				$ gem install rails

objects

	php	perl	python	ruby

define class	class Int {
	  private $value;
	  function __construct($int=0) {
	  $this->value = $int;
	  }
	  function getValue() {
	  return $this->value;
	  }
	  function setValue($i) {
	  $this->value = $i;
	  }
	}
		package Int;
		use Moose;
		has 'value' => (is => 'rw');
		around BUILDARGS => sub {
		  my $orig = shift;
		  my $class = shift;
		  my $v = $_[0] || 0;
		  $class->$orig(value => $v);
		};
		no Moose;
			class Int:
			  def __init__(self, v=0):
			  self.value = v
				class Int
				  attr_accessor :value
				  def initialize(i=0)
				  @value = i
				  end
				end

create object	$i = new Int();	my $i = new Int(); # or my $i = Int->new();	i = Int()	i = Int.new
invoke getter, setter	$v = $i->getValue(); $i->setValue($v+1);	my $v = $i->value; $i->value($v+1);	v = i.value i.value = v+1	v = i.value i.value = v+1
instance variable accessibility	*must be declared*	*private by default*	*public; attributes starting with underscore private by convention*	*private by default; use* attr_reader, attr_writer, attr_accessor *to make public*

define method	function plus($i) {
	  return $this->value + $i;
	}
		# in package:
		sub plus {
		  my $self = shift;
		  $self->value + $_[0];
		}
			def plus(self,v):
			  return self.value + v
				def plus(i)
				  value + i
				end

invoke method	$i->plus(7)	$i->plus(7)	i.plus(7)	i.plus(7)

destructor	function __destruct() {
	  echo "bye, $this->value\n";
	}
		# in package:
		sub DEMOLISH {
		  my $self = shift;
		  my $v = $self->value;
		  print "bye, $v\n";
		}
			def __del__(self):
			  print("bye, %", self.value)
				val = i.value
				ObjectSpace.define_finalizer(int) {
				  puts "bye, #{val}"
				}

method missing	function __call($name, $args) {
	  $argc = count($args);
	  echo "no def: $name " .
	  "arity: $argc\n";
	}
		# in package:
		our $AUTOLOAD;
		sub AUTOLOAD {
		  my $self = shift;
		  my $argc = scalar(@_);
		  print "no def: $AUTOLOAD"
			def __getattr__(self, name):
			  s = "no def: "+name+" arity: %d"
			  return lambda *a: print(s % len(a))
				def method_missing(name, *a)
				  puts "no def: #{name}" +
				  " arity: #{a.size}"
				end

	} }	. " arity: $argc\n"; }		

inheritance	class Counter extends Int {
	private static $instances = 0;
	function __construct($int=0) {
	Counter::$instances += 1;
	parent::__construct($int);
	}
	function incr() {
	$i = $this->getValue();
	$this->setValue($i+1);
	}
	static function getInstances() {
	return $instances;
	}
	}
		package Counter;
		use Moose;
		extends 'Int';
		my $instances = 0;
		sub BUILD {
		$instances += 1;
		}
		sub incr {
		my $self = shift;
		my $v = $self->value;
		$self->value($v + 1);
		}
		sub instances {
		$instances;
		}
		no Moose;
			class Counter(Int):
			instances = 0
			def __init__(self, v=0):
			Counter.instances += 1
			Int.__init__(self, v)
			def incr(self):
			self.value += 1
				class Counter < Int
				@@instances = 0
				def initialize
				@@instances += 1
				super
				end
				def incr
				self.value += 1
				end
				def self.instances
				@@instances
				end
				end

invoke class method	Counter::getInstances()	Counter::instances();	Counter.instances	Counter.instances

reflection and hooks

	php	perl	python	ruby
class	get_class($a)	ref $a	type(a)	a.class
has method?	method_exists($a, 'reverse')	$a->can('reverse')	hasattr(a,'reverse')	a.respond_to?('reverse')

message passing	for ($i = 1; $i <= 10; $i++) {
	call_user_func(array($a,
	"phone$i"), NULL);
	}
		for $i (0..10) {
		$meth = "phone$i";
		$a->$meth(undef);
		}
			for i in range(1,10):
			getattr(a,"phone"+str(i))(None)
				(1..9).each do |i|
				a.send("phone#{i}="nil)
				}

eval		while(<>) {
		print ((eval), "\n");
		}
			while True:
			print(eval(sys.stdin.readline()))
				loop do
				puts eval(gets)
				end

methods		$class = ref($a); keys eval "%${class}::";	[m for m in dir(a) if callable(getattr(a,m))]	a.methods
attributes		keys %$a;	dir(a)	a.instance_variables
pretty print	$h = array('foo'=>1, 'bar'=>array(2,3)); print_r($h);	require 'dumpvar.pl'; %h = ('foo'=>1, 'bar'=>[2, 3]); dumpValue(\%h);	import pprint h = {'foo':1, 'bar':[2,3] } pprint.PrettyPrinter().pprint(h)	require 'pp' h = { 'foo'=>1, 'bar'=>[2,3] } pp h
source line number and file name	__LINE__ __FILE__	__LINE__ __FILE__	import inspect c = inspect.currentframe() c.f_lineno	__LINE__ __FILE__

			c.f_code.co_filename	

web

	php	perl	python	ruby
http get	$h = 'http://www.google.com'; $ch = curl_init($h); curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1); $output = curl_exec($ch); curl_close($ch); $output	require HTTP::Request; require LWP::UserAgent; $h = 'http://www.google.com'; $r = HTTP::Request- >new(GET=>$h); $ua = LWP::UserAgent->new; $resp = $ua->request($r); $resp->content()	import httplib h = 'www.google.com' f = httplib.HTTPConnection(h) f.request("GET",'/') f.getresponse().read()	require 'net/http' h = 'www.google.com' r = Net::HTTP.start(h, 80) do |f| f.get('/') end r.body
url encode/decode	urlencode("hello world"); urldecode("hello+world");	use CGI; CGI::escape('hello world') CGI::unescape('hello+world')	import urllib urllib.quote_plus("hello world") urllib.unquote_plus("hello+world")	require 'cgi' CGI::escape('hello world'); CGI::unescape('hello+world');
build xml	$x = '<a>'; $d = new SimpleXMLElement($x); $d->addChild('b', 'foo'); echo $d->asXML();		# not ported to python 3 from xmlbuilder import XMLBuilder builder = XMLBuilder() with builder.a: with builder.b: builder << "foo" print(str(builder))	# gem install builder require 'builder' builder = Builder::XmlMarkup.new xml = builder.a do |child| child.b("foo") end puts xml

parse xml	$x = '<a>foo';
	$d = simplexml_load_string($x);
	foreach ($d->children() as $c) {
	break;
	}
	echo $c;
		# download XML::Simple from CPAN:
		use XML::Simple;
		$xml = new XML::Simple;
		$s = '<a>foo';
		$doc = $xml->XMLin($s);
		print $doc->{'b'};
			from xml.etree import ElementTree
			xml = '<a>foo'
			doc = ElementTree.fromstring(xml)
			print(doc[0].text)
				require 'rexml/document'
				xml = '<a>foo'
				doc = REXML::Document.new(xml)
				puts doc[0][0].text

xpath	$x = '<a><c>foo</c>'; $d = simplexml_load_string($x); $n = $d->xpath('/a/b/c'); echo $n[0];		from xml.etree import ElementTree xml = '<a><c>foo</c>' doc = ElementTree.fromstring(xml) node = doc.find("b/c") print(node.text)	require 'rexml/document' include REXML xml = '<a><c>foo</c>' doc = Document.new(xml) node = XPath.first(doc,'/a/b/c') puts node.text
json	$a = array('t'=>1, 'f'=>0); $s = json_encode($a); $h = json_decode($s, TRUE);	# download JSON from CPAN: use JSON; $raw = { 't' => 1, 'f' => 0 }; $json = JSON->new- >allow_nonref; $s = $json->encode($raw); $h = $json->decode($s);	import json s = json.dumps({'t':1, 'f':0}) h = json.loads(s)	require 'json' s = {'t'=> 1,'f'=> 0}.to_json h = JSON.parse(s)

web framework

	php	perl	python	ruby
web framework				

web framework			*Django 1.2.5*	*Rails 3.0.1*

create project and start server			$ django-admin.py startproject foo
			$ cd foo
			$ python manage.py runserver 4444
			$ wget localhost:4444
				$ rails new foo -d mysql
				$ cd foo
				$ rails server -p 3333 &
				$ wget localhost:3333

console			$ python manage.py shell	$ rails console

create view			$ python manage.py startapp bar
			$ cat <<EOF >bar/views.py
			from django.shortcuts import *
			def baz(request):
			  data = {}
			  return render_to_response(
			  'baz.html', data)
			EOF
			$ mkdir templates
			$ cat <<EOF >templates/baz.html
			 Baz
			EOF
			in urls.py *import* bar.views *and add this tuple to* patterns:
			(r'bar/baz', bar.views.baz)
			in settings.py *add* 'bar' *to* INSTALLED_APPS. *Add the full pathname to the* templates *directory to* TEMPLATE_DIRS.
				$ rails generate controller bar baz
				$ wget localhost:3333/bar/baz
				controller is method baz *in* app/controllers/bar_controller.rb
				$ cat <<EOF >app/views/baz/baz.html.erb
				 Baz
				EOF

url routing file			urls.py	config/routes.rb
display url routes				rake routes
routing a url to a view				
extract parameter from route				
reverse lookup				

database setup			$ cat <<EOF >db.sql
			create database foo;
			EOF
			$ mysql < db.sql
				$ cat <<EOF >db.sql
				create database foo_development;
				create database foo_test;
				EOF
				$ mysql < db.sql

db config file			*settings.py*	*config/database.yml*

create model			$ cat <<EOF >>bar/models.py
			class Customer(models.Model):
			  name = models.CharField(max_length=200)
				$ cat <<EOF >db/customers.sql
				create table customers (
				  id serial,
				  name varchar(200)
				);
				EOF

			EOF
			$ python manage.py syncdb
				$ mysql foo_development
				> source db/customers.sql
				> exit
				$ rails generate model customer

migration				*edit file in* db/migrate
				$ rake db:drop
				$ rake db:create
				$ rake db:migrate

create row			from bar.models import Customer c1 = Customer(name='Dan Foo') c1.save()	h1 = {:name=>'Dan Foo'} c1 = Customer.create!(h1) h2 = {:name=>'Don Bar'} c2 = Customer.new(h2) c2.save!
find row			from bar.models import Customer c = Customer.objects.get(name='Dan Foo')	n = 'Dan Foo' c = Customer.find_by_name(n) c = Customer.find(:first, :conditions => ['name = ?', n])
templates	*PHP*			*ERB*
making data available to templates				

java interoperation				
	php	**perl**	**python**	**ruby**
version			Jython 2.5.1	JRuby 1.4.0
repl			jython	jirb
interpreter			jython	jruby
compiler			*none in 2.5.1*	jrubyc
prologue			import java	*none*
new			rnd = java.util.Random()	rnd = java.util.Random.new
method			rnd.nextFloat()	rnd.next_float
import			from java.util import Random rnd = Random()	java_import java.util.Random rnd = Random.new
non-bundled java libraries			import sys sys.path.append('path/to/mycode.jar')	require 'path/to/mycode.jar'

			import MyClass	
shadowing avoidance			import java.io as javaio	module JavaIO include_package "java.io" end
to java array			import jarray jarray.array([1,2,3],'i')	[1,2,3].to_java(Java::int)
java class subclassable?			*yes*	*yes*
java class open?			*no*	*yes*

General

versions used

The versions used for testing code in the cheat sheet.

show version

How to get the version.

php:

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

perl:

Also available in the predefined variable $], or in a different format in $^V and $PERL_VERSION.

python:

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

```
import platform
platform.python_version()
```

ruby:

Also available in the global constant *VERSION* (ruby 1.8.7) or *RUBY_VERSION* (ruby 1.9.1).

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:

Submit each line of code to *php -a* as <?= *php code* ?> to have the line executed as php code and the result displayed.

perl:

The perl repl lacks readline and does not save or display the result of an expression. Actually, 'perl -d' runs the perl debugger, and 'perl -e' runs code provided on the command line.

python:

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

ruby:

irb saves the result of the last statement in _.

check syntax

How to check the syntax of code without executing it.

flags for stronger and strongest warnings

Flags to increase the warnings issued by the interpreter:

python:

The -t flag warns about inconsistent use of tabs in the source code. The -3 flag is a Python 2.X option which warns about syntax which is no longer valid in Python 3.X.

statement separator

How the parser determines the end of a statement.

php:

The ; is a separator and not a terminator. It is not required before a closing ?> tag of a php block.

perl:

In a script statements are separated by semicolons and never by newlines. However, when using 'perl -de 0', a newline terminates the statement.

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.

block delimiters

How blocks are delimited.

perl:

Curly brackets {} delimit blocks. They are also used for:

- hash literal syntax which returns a reference to the hash: $rh = { 'true' => 1, 'false' => 0 }
- hash value lookup: $h{'true'}, $rh->{'true'}

- variable name delimiter: $s = "hello"; print "${s}goodbye";

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 encouters 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.

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.

assignment

How to assign a value to a variable.

perl:

Assignment operators have right precedence and evaluate to the right argument, so assignments can be chained:

```
$a = $b = 3;
```

python:

If the variable on the left does not exist, then it is created.

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 (=) in place of an equality test (==). Assignments can nevertheless be chained to assign multiple variables to the same value:

```
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
```

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 value of the variable as an argument.

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 compound 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

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--;
```

perl:

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.

local variable declarations

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*.

perl:

Variables don't need to be declared unless *use strict* is in effect.

If not initialized, scalars are set to *undef*, arrays are set to an empty array, and hashes are set to an empty hash.

Perl can also declare variables with *local*. These replace the value of a global variable with the same name, if any, for the duration of the enclosing scope, after which the old value is restored. *local* declarations became obsolete with the introduction of the *my* declaration introduced in Perl 5.

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 local scope

A list of regions which define a 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 from being placed inside another scope defining region.

A region is *nestable* if it can be placed inside another cope 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.

perl:

A local variable can be defined outside of any function definition or anonymous block, in which case the scope of the variable is the file containing the source code. In this way Perl resembles Ruby and contrasts with PHP and Python. In PHP and Python, any variable defined outside a function definition is global.

In Perl, when a region which defines a scope is nested inside another, then the inner region has read and write access to locql variables defined in the outer region.

Note that the blocks associated with the keywords *if*, *unless*, *while*, *until*, *for*, and *foreach* are anonymous blocks, and thus any *my* declarations in them create variables local to the block.

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.

ruby:

Note that though the keywords *if*, *unless*, *case*, *while*, and *until* each define a block which is terminated by an *end* keyword, 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
```

In Ruby 1.8, the scope of the parameter of an anonymous block or function or block is local to the block or function body if the name is not already bound to a variable in the containing scope. However, if it is, then the variable in the containing scope will be used. This behavior was changed in Ruby 1.9 so that parameters are always local to function body or block. Here is an example of code which behaves differently under Ruby 1.8 and Ruby 1.9:

```
x = 3
id = lambda { |x| x }
id.call(7)
puts x # 1.8 prints 7; 1.9 prints 3
```

Ruby 1.9 also adds the ability mark variables as local, even when they are already defined in the containing scope. All such variables are listed inside the parameter pipes, separated from the parameters by a semicolon:

```
x = 3
noop = lambda { |; x| x = 15 } # bad syntax under 1.8
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.

perl:

Undeclared variables, which are permitted unless *use strict* is in effect, are global. If *use strict* is in effect, a global can be declared at the top level of a package (i.e. outside any blocks or functions) with the *our* keyword. A variable declared with *my* inside a function will hide a global with the same name, if there is one.

python:

A variable is 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 individuality 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: $.

static variable

How to create a variable which is private to a function but keeps state between invocations.

constant declaration

How to declare a constant.

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.

to-end-of-line comment

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

multiline comment

How to comment out multiple lines.

python:

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

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"; }
```

perl:

$v == *undef* does not imply that *$v* is *undef*. Any comparison between *undef* and a falsehood will return true. The following comparisons are true:

[code]]
$v = undef;
if ($v == undef) { print "true"; }
$v = 0;
if ($v == undef) { print "sadly true"; }
$v = '';
if ($v == undef) { print "sadly true"; }
[[/code]]

undefined variable access

The result of attempting to access an undefined variable.

undefined test

Perl does not distinguish between unset variables and variables that have been set to undef. In perl, calling defined($a) does not result in a error if $a is undefined, even with the strict pragma.

Arithmetic and Logic

true and false

php:

Any identifier which matches TRUE case-insensitive can be used for the TRUE boolean. Similarly for FALSE. Note that variable names are case-sensitive, but function names are case-insensitive.

falsehoods

python:

Whether a object evaluates to True or False in a boolean context can be customized by implementing a __nonzero__ instance method for the class.

logical operators

Logical and, or, and not.

php:

&& || and ! have higher precedence than assignment and the ternary operator (?:). *and*, *or*, and *xor* have lower precedence than assignment and the ternary operator.

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.

comparison operators

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

php:

Most of the comparison 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.

perl:

The operators: == != > < >= <= convert strings to numbers before performing a comparison. Many string evaluate as zero in a numeric context and are equal according to the == operator. To perform a lexicographic string comparison, use: *eq*, *ne*, *gt*, *lt*, *ge*, *le*.

convert from string, to string

How to convert string data to numeric data and vice versa.

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.

perl:

Perl 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.

arithmetic operators

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

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)
```

integer division

How to perform division which returns a integer.

float division

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

arithmetic functions

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

perl:

```
sub tan { sin($_[0]) / cos($_[0]); }
sub asin { atan2( sqrt(1-$_[0]*$_[0]), $_[0]); }  # angle from 0 to pi radians
sub acos { atan2( $_[0], sqrt(1-$_[0]*$_[0])); } # angle from -pi/2 to pi/2
```

arithmetic truncation

How to take the absolute value; 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.

perl:

Here is an implementation of *round* in perl which handles negative numbers correctly:

```
sub round {
    my($number) = shift;
    return int($number + .5 * ($number <=> 0));
}
```

To get the *ceil* and *floor* routines, put this at the top of the script:

```
use POSIX qw(ceil floor);
```

min and max

How to get the min and max.

division by zero

What happens when division by zero is performed.

integer overflow

What happens when the largest representable integer is exceeded.

float overflow

What happens when the largest representable float is exceeded.

sqrt -2

The result of taking the square root of negative two.

rational numbers

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

perl:

Rational number modules are available on CPAN.

ruby:

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

complex numbers

perl:

Complex number modules are available on CPAN.

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.

bit operators

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

Strings

character literal

Most of the languages in this sheet manipulate characters as integers or strings containing a single character.

chr and ord

converting characters to ASCII codes and back

string literal

The syntax for string literals.

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"""
```

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.

perl:

Put the custom identifer in single quotes to prevent variable interpolation and backslash escape interpretation:

```
s = <<'EOF';
```

```
Perl code uses variables with dollar
signs, e.g. $var
EOF
```

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 variable into strings.
EOF
```

string escapes

python:

Before python 3, it was possible to enter a character by Unicode point using the following syntax:

```
u'\u03bb'
```

encoding

variable interpolation

How to interpolate variables into strings.

python:

Python lacks interpolating quotes. Except for the fact that they can contain single quotes, double quotes are identical to single quotes.

string length

character count

how to count the number times a character occurs in a string

perl:

The translate characters *tr* operator returns the number of characters replaced. Normally it performs a destructive operation on the string, but not if the second half of the translation specification is empty.

index of substring

extract substring

string concatenate

The string concatenation operator.

split

ruby:

The argument to split can be a string or regular expression. A 2nd argument will put a limit on the number of elements in the array; the last element will contain the reaminder of the string. To split a string into an array of single character strings, use

```
"abcdefg".split('')
```

join

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

scan

perl:

The Perl module *String::Scanf*, which is not part of the standard distribution, has a *sscanf* function.

pack and unpack

sprintf

How to create a string using a printf style format.

python:

Escape curly braces by doubling:

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

case manipulation

strip

pad on right, on left

character translation

regexp match

How to test whether a string matches a regular expression.

match, prematch, postmatch

substring matches

single substitution

perl:

The =~ operator performs the substitution in place on the string, and returns the number of substitutions performed.

python:

The 3rd argument of *sub* indicates the maximum number of substitutions to be performed. The default value is zero, in which case there is no limit to the number of substitutions performed.

ruby:

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

global substitution

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

perl:

The =~ operator performs the substitution in place on the string, and returns the number of substitutions performed.

python:

To perform a single substitution use

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.

```
re.compile('teh').sub('the', 'teh last of teh Mohicans',1)
```

ruby:

Use *sub* to perform a single substitution.

Containers

Although the scripting languages have the same basic container types, terminology is not universal:

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

php:

PHP uses the same data structure for arrays and hashes.

perl:

array refers to a data type. *list* refers to a context.

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.

array literal

Array literal syntax.

perl:

Square brackets create an array and return a reference to it:

```
$a = [1,2,3]
```

array size

How to get the number of elements in an array.

perl:

The idiomatic way to test whether an array is empty in perl:

```
if ( @a ) {
  print "@a is empty\n";
}
```

python:

The idiomatic way to test whether a list is empty in python:

```
if a:
  print "a is empty"
```

ruby:

The empty list evaluates to true in a condition test. Use the empty? predicate to determine if an array is empty:

```
if a.empty?
  puts "a is empty"
end
```

array lookup

How to access a value in an array by index.

perl:

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

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.

index of array element

perl:

Some techniques for getting the index of an array element.

array slice

How to slice a subarray from an array.

python:

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

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

Python has notation for taking every nth element:

```
>>> nums=[1,2,3,4,5]
```

```
>>> nums[0::2]
[1, 3, 5]
```

ruby:

There is notation for specifying the first index and the number of items in the slice

```
irb> nums = [1,2,3,4,5]
=> [1, 2, 3, 4, 5]
irb> nums[1,3]
=> [2, 3, 4]
```

manipulate back of array

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

manipulate front of array

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

array concatenation

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

address copy, shallow copy, deep 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.

perl:

Taking a reference is customary way to make an address copy in Perl, but the Perl example is not equivalent to the other languages because it makes @$b and @a refer to the same array. To make @b and @a refer to the same array, use typeglobs:

```
*b = *a;
```

The Clone module, available from CPAN, can be used to make deep copies:

```
use Clone qw(clone);
$d = clone(\@a);
```

python:

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

```
c = a[:]
```

arrays as function arguments

How arrays are passed as arguments.

array iteration

How to iterate through the elements of an array.

indexed array iteration

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

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.

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.

membership

How to test for membership in an array.

intersection

How to compute an intersection.

python:

Python has a set data type which is built-in as of version 2.4. Furthermore, Python has literal syntax for sets: {1,2,3}.

The example computes the intersection by converting the lists to sets. The return value is a set; it can be converted to a list with the *list* constructor.

ruby:

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

union

python:

The python example computes the union by converting the lists to sets. The return value is a set; it can be converted to a list with the *list* constructor.

ruby:

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

map

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

ruby:

The *map!* method will apply the function to the elements of the source array in place.

filter

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

reduce

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

perl:

The *reduce* function was added in Perl 5.10.

universal test

How to test whether a condition holds for all members of an array. Always true for an empty array.

A universal test can be implemented with a filter: filter the set on the negation of the predicate and test whether the result is empty.

existential test

How to test whether an item in an array exists for which a condition holds. Always false for an empty array.

An existential test can be implemented with a filter: filter the set on the predicate and test whether the result is non-empty.

dictionary literal

perl:

Curly brackets create a hash and return a reference to it:

```
$h = { 'hello' => 5, 'goodbye' => 7 }
```

dictionary lookup

How to lookup a dictionary value using a dictionary key.

is dictionary 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.

dictionary size

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

dicionary iteration

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

python:

Pre-3.0 versions of python iterate through the key value pairs in a dictionary with *iteritems()* instead of *items()*.

keys and values of dictionary as array

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

out of bounds behavior

What happens when a lookup is performed using an index that is not in the range of an array; what happens when a lookup is performed on a key that is not in a dictionary.

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.

function declaration

function invocation

python:

When invoking methods and functions, 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.

Starting with 3.0, print is treated as a function instead of a keyword. Thus parens are mandatory around the print 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. As of Ruby 1.9, it is mandatory that the left paren not be separated from the method name by whitespace.

missing argument behavior

How incorrect number of arguments upon invocation are handled.

perl:

Perl collects all arguments into the @_ array, and subroutines normally don't declare the number of arguments they expect. However, this can be done with prototypes. Prototypes also provide a method for taking an array from the caller and giving a reference to the array to the callee.

python:

TypeError is raised if the number of arguments is incorrect.

ruby:

ArgumentError is raised if the number of arguments is incorrect.

default value

How to declare a default value for an argument.

arbitrary number of arguments

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

php:

It is also possible to use *func_num_args* and *func_get_arg* to access the arguments:

```
for ($i = 1; $i < func_num_args(); $i++) {
    echo func_get_arg($i);
}
```

named parameters

How to write a function which uses named parameters.

python:

In a function definition, 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.

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

ruby:

pass number or string by reference

perl:

Scalars are passed by reference.

pass array or hash by reference

perl:

Arrays and hashes are not passed by reference by default. If an array is provided as a argument, each element of the array will be assigned to a parameter. A change to the parameter will change the corresponding value in the original array, but the number of elements in the array cannot be increased. To write a function which changes the size of the array the array must be passed by reference using the backslash notation.

When a hash is provided as a argument each key of the has will be assigned to a parameter and each value of the hash will be assigned to a parameter. In other words the number of parameters seen by the body of the function will be twice the size of the hash. Each value parameter will immediately follow its key parameter.

return value

lambda declaration

How to define 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.

lambda invocation

function with private state

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

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 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.

generator

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

python:

Python generators are typically used in *for/in* statements and list comprehensions.

ruby:

Ruby generators are called fibers. Fibers are coroutines.

Execution Control

if

Some optional branching constructs:

perl:

A switch statement was introduced with perl 5.10.

ruby:

```
case a:
when 0:
  puts "no"
when 1:
  puts "yes"
when 2:
  puts "maybe"
else
  puts "error"
end
```

while

ruby:

loop creates a loop with no exit condition. *until* exits when the condition is true.

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.

for

How to write a C-style for loop.

range iteration

How to iterate over a range of integers.

php:

The *range* operator creates an array in memory. Hence a for loop is more efficient for the given example.

perl:

In a array context such as exists inside the parens of the *for* construct, the .. operator creates an array in memory. Hence a C-style for loop is more efficient for the given example.

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
```

global variable for last exception raised

The global variable name for the last exception raised.

define exception

How to define a new variable class.

catch exception by type and assign to variable

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.

uncaught exception behavior

System behavior if an exception goes uncaught. Most interpreters print the exception message to stderr and exit with a nonzero status.

start thread

ruby:

Ruby 1.8 threads are green threads, and the interpreter is limited to a single operating system thread.

wait on thread

Environment and I/O

external command

When using tclsh as a shell or repl, external commands that do not have the same name as a built-in or user defined function can be executed directly without using the exec command.

backticks

command line args

print to standard out

python:

print appends a newline to the output. To suppress this behavior, put a trailing comma after the last argument. If given multiple arguments, *print* joins them with spaces.

ruby:

puts appends a newline to the output. *print* does not.

standard file handles

open file

ruby:

When *File.open* is given a block, the file is closed when the block terminates.

open file for writing

close file

read line

iterate over a file by line

chomp

Remove a newline, carriage return, or carriage return newline pair from the end of a line if there is one.

php:

chop removes all trailing whitespace. It is an alias for *rtrim*.

perl:

chomp modifies its argument, which thus must be a scalar, not a string literal.

python:

Python strings are immutable. *rstrip* returns a modified copy of the string. *rstrip('\r\n')* is not identical to *chomp* because it removes all contiguous carriage returns and newlines at the end of the string.

ruby:

chomp! modifies the string in place. *chomp* returns a modified copy.

read entire file

How to read the contents of a file into memory.

write to file

How to write to a file handle.

flush file

How to flush a file handle that has been written to.

environment variable

exit

python:

It is possible to register code to be executed upon exit:

```
import atexit
atexit.register(print, "goodbye")
```

It is possible to terminate a script without executing registered exit code by calling *os._exit*.

ruby:

It is possible to register code to be executed upon exit:

```
at_exit { puts "goodbye" }
```

The script can be terminated without executing registered exit code by calling *exit!*.

set signal handler

Libraries and Modules

load library

Execute the specified file. Normally this is used on a file which only contains declarations at the top level.

php:

include_once behaves like *require_once* except that it is not fatal if an error is encountered executing the library.

If it is desirable to reload the library even if it might already have been loaded, use *require* or *include*.

perl:

The last expression in a perl library must evaluate to true. When loading a library with *use*, the suffix of the file must be *.pm*.

The *do* directive will re-execute a library even if it has already been loaded.

library path

module declaration

module separator

package management

How to show the installed 3rd party packages, and how to install a new 3rd party package.

Objects

define class

php:

Properties (i.e. instance variables) must be declared *public*, *protected*, or *private*. Methods can optionally be declared *public*, *protected*, or *private*. Methods without a visibility modifier are public.

perl:

The cheatsheet shows how to create objects using the CPAN module Moose. To the client, a Moose object behaves like a traditional Perl object; it's when a class needs to be defined that Moose code looks different from traditional Perl code.

Moose provides these additional features:

- shortcuts for creating accessors and delegates
- attributes can be declared to a be a type
- classes are objects and can be reflected upon
- mixins, which are called *roles*

Moose objects are illustrated instead of Perl objects because Moose is used by the Catalyst web framework, and some of the features in Moose are in the Perl 6 object system. Here is how to define a class in the traditional Perl way:

```
package Int;

sub new {
  my $class = shift;
  my $v = $_[0] || 0;
  my $self = {value => $v};
  bless $self, $class;
  $self;
}

sub value {
```

```
  my $self = shift;
  if ( @_ > 0 ) {
    $self->{'value'} = shift;
  }
  $self->{'value'};
}

sub add {
  my $self = shift;
  $self->value + $_[0];
}

sub DESTROY {
  my $self = shift;
  my $v = $self->value;
  print "bye, $v\n";
}
```

python:

As of Python 2.2, classes are of two types: new-style classes and old-style classes. The class type is determined by the type of class(es) the class inherits from. If no superclasses are specified, then the class is old-style. As of Python 3.0, all classes are new-style.

New-style classes have these features which old-style classes don't:

- universal base class called *object*.
- descriptors and properties. Also the __getattribute__ method for intercepting all attribute access.
- change in how the diamond problem is handled. If a class inherits from multiple parents which in turn inherit from a common grandparent, then when checking for an attribute or method, all parents will be checked before the grandparent.

create object

invoke getter, setter

perl:

Other getters:

```
$i->value()
$i->{'value'}
```

Other setters:

```
$i->{'value'} = $v;
```

python:

Defining explicit setters and getters in Python is considered poor style. If it becomes necessary to extra logic to attribute, this can be achieved without disrupting the clients of the class by creating a property:

```
def getValue(self):
  print("getValue called")
  return self.__dict__['value']
def setValue(self,v):
  print("setValue called")
  self.__dict__['value'] = v
value = property(fget=getValue, fset = setValue)
```

instance variable accessibility

define method

invoke method

perl:

If the method does not take any arguments, the parens are not necessary to invoke the method.

destructor

perl:

Perl destructors are called when the garbage collector reclaims the memory for an object, not when all references to the object go out of scope. In traditional Perl OO, the destructor is named *DESTROY*, but in Moose OO it is named *DEMOLISH*.

python:

Python destructors are called when the garbage collector reclaims the memory for an object, not when all references to the object go out of scope.

ruby:

Ruby lacks a destructor. It is possible to register a block to be executed before the memory for an object is released by the garbage collector. A ruby interpreter may exit without releasing memory for objects that have gone out of scope and in this case the finalizer will not get called. Furthermore, if the finalizer block holds on to a reference to the object, it will prevent the garbage collector from freeing the object.

method missing

php:

Define the method *__callStatic* to handle calls to undefined class methods.

python:

__getattr__ is invoked when an attribute (instance variable or method) is missing. By contrast, *__getattribute__*, which is only available in Python 3, is always invoked, and can be used to intercept access to attributes that exist. *__setattr__* and *__delattr__* are invoked when attempting to set or delete attributes that don't exist. The *del* statement is used to delete an attribute.

ruby:

Define the method *self.method_missing* to handle calls to undefined class methods.

inheritance

perl:

Here is how inheritance is handled in traditional Perl OO:

```
package Counter;

our @ISA = "Int";

my $instances = 0;
our $AUTOLOAD;

sub new {
  my $class = shift;
  my $self = Int->new(@_);
  $instances += 1;
  bless $self, $class;
  $self;
}

sub incr {
  my $self = shift;
  $self->value($self->value + 1);
}
sub instances {
  $instances;
}

sub AUTOLOAD {
  my $self = shift;
  my $argc = scalar(@_);
  print "undefined: $AUTOLOAD " .
```

```
    "arity: $argc\n";
}
```

invoke class method

Reflection and Hooks

class

has method?

perl:

$a->can() returns a reference to the method if it exists, otherwise it returns *undef*/.

python:

hasattr(o,'reverse') will return *True* if there is an instance variable named 'reverse'.

message passing

eval

methods

perl:

The following code

```
$class = ref($a);
keys eval "%${class}::";
```

gets all symbols defined in the namespace of the class of which *$a* is an instance. The *can* method can be used to narrow the list to instance methods.

attributes

perl:

keys %$a assumes the blessed object is a hash reference.

python:

dir(o) returns methods and instance variables.

pretty print

How to display the contents of a data structure for debugging purposes.

source line number and file name

How to get the current line number and file name of the source code.

Web

http get

url encode/decode

build xml

parse xml

xpath

json

Web Framework

web framework

python:

Pylons 1.0 only works with Python 2. I was not able to install it on my Mac, so I run it under Linux. I accepted the default choices for templating engine (Mako) and ORM (SQLAlchemy).

create project and start server

python:

To make Django aware of the newly created `bar` app, edit the file `settings.py` adding `'bar'` to the `INSTALLED_APPS` list.

rails:

If the database is not specified when the repository is created it defaults to sqlite3.

console

create view

python:

The fact that `TEMPLATE_DIRS` in `settings.py` must contain full pathnames will cause problems. Putting the following at the top of the `settings.py` file is advised:

```
import os

def relative_path(relative_path):
    root = os.path.dirname(os.path.abspath(__file__))
    return os.path.join(root, relative_path)
```

The pathnames can then be specified in `TEMPLATE_DIRS` relative to the directory that `settings.py` is in:

```
    relative_path('templates')
```

object relational mapper

database setup

python:

ruby:

If the rails project was created with *-d mysql* and the databases names are *project_name*_development and *project_name*_test, then it is not necessary to edit config/databases.yml to connect to the database.

db config file

Name and location of the database configuration file.

create model

migration

ruby:

The *rails generate model* command creates a migration file in *db/migrate*. The file can be edited to add columns which rails does not create by default. In the file below the line creating the *name* column is the only addition:

```
class CreateCustomers < ActiveRecord::Migration
  def self.up
    create_table :customers do |t|
      t.string :name
      t.timestamps
    end
  end

  def self.down
    drop_table :customers
  end
end
```

create row

find row

templates

Java Interoperation

version

prologue

Code necessary to make java code accessible.

new

How to create a java object.

method

How to invoke a java method.

import

How to import names into the current namespace.

PHP

PHP Manual
General Style and Syntax Codeigniter
Coding Standards Pear
PHP Style Guide Apache

Perl

perldoc
man perlstyle

The first character of a perl variable ($, @, %) determines the type of value that can be stored in the variable (scalar, array, hash). Using an array variable (@foo) in a scalar context yields the size of the array, and assigning scalar to an array will set the array to contain a single element. $foo[0] accesses the first element of the array @foo, and $bar{'hello'} accesses the value stored under 'hello' in the hash %bar. $#foo is the index of the last element in the array @foo.

Scalars can store a string, integer, or float. If an operator is invoked on a scalar which contains an incorrect data type, perl will always perform an implicit conversion to the correct data type: non-numeric strings evaluate to zero.

Scalars can also contain a reference to a variable, which can be created with a backslash: $baz = \@foo; The original value can be dereferenced with the correct prefix: @$baz. References are how perl creates complex data structures, such as arrays of hashes and arrays of arrays. If $baz contains a reference to an array, then $baz->[0] is the first element of the array. if $baz contains a reference to a hash, $baz->{'hello'} is the value indexed by 'hello'.

The literals for arrays and hashes are parens with comma separated elements. Hash literals must contain an even number of elements, and '=>' can be used in placed of a comma ',' between a key and its value. Square brackets, e.g. [1, 2, 3], create an array and return a reference to it, and curly brackets, e.g. { 'hello' => 5, 'bye' => 3 }, create a hash and return a reference to it.

By default perl variables are global. They can be made local to the containing block with the my or the local keyword. my gives lexical scope, and local gives dynamic scope. Also by default, the perl interpreter creates a variable whenever it encounters a new variable name in the code. The 'use strict;' pragma requires that all variables be declared with my, local, or our. The last is used to declare global variables.

perl functions do not declare their arguments. Any arguments passed to the function are available in the @_ array, and the shift command will operate on this array if no argument is specified. An array passed as an argument is expanded: if the array contains 10 elements, the callee will have 10 arguments in its @_ array. A reference (passing \@foo instead of @foo) can be used to prevent this.

Some of perl's special variables:

- $$: pid of the perl process
- $0: name of the file containing the perl script (may be a full pathname)
- $@: error message from last eval or require command
- $&, $`, $': what last regexp matched, part of the string before and after the match
- $1..$9: what subpatterns in last regexp matched

Python

2.6
Why Python3 Summary of Backwardly Non-compatible Changes in Python 3
3.1
Jython 2.5.1
Jython Wiki
PEP 8: Style Guide for Python Code van Rossum

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}. 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

1.8.7 core
1.9 core
1.8.6 stdlib
Ruby Style Neukirchen
The Unofficial Ruby Usage Guide Macdonald

Ruby consistently treats all values as objects. Classes are objects. Methods, however, are not objects. The system provided classes are open: i.e. the user can add methods to String, Array, or Fixnum. Another difference from python (and perl) is that ruby only permits single inheritance. However, 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.

In ruby, there is no difference between named functions and methods: top level 'functions' are in fact methods defined on the main object. All methods have a receiver which can be referenced with the self: the receiver does not needed to be treated as an argument like in perl and python.

Ruby has syntax which aids functional programming: a ruby method invocation can be followed by a block, which is a closure. The invoked method can call the block with yield(arg1,...). Also, the invoked method can store the block in a variable if its signature ends with &<varname>. The block can then be invoked with <varname>.call(arg1,...).

History

A History of Easy-to-Use Languages

Easy-to-use languages make use of a number of unrelated innovations, each with their own history.

high level languages

Most computers have register based architectures. The native language is a set of instructions which operate on arguments in specific registers.

In Fortran (1956) values could be stored in named variables and the programmer does not need to worry about where the value is stored. The variable names can be combined in arithmetic expressions using the customary notation of mathematics, though the asterisk (*) is used for multiplication instead of adjacency. Variables names consist of a letter following by zero to five letters or digits. The type of a variable could be integer or floating point and was determined by the initial letter: variable names starting with I, J, K, L, M, or N are integers.

For execution control, Fortran used statement labels, a GOTO instruction, and an IF statement. The IF statement consisted of an algebraic expression and 3 statement labels; it branched to one of the labels depending upon whether the algebraic expression was less than, equal to, or greater than zero.

Fortran II (1958) added user-defined functions, perhaps the most important abstraction mechanism available to programmers even today. It also supported separate compilation.

Algol 58 (1958) introduced type declarations for variables, which could be boolean or integer. It added logical operators and comparison operators. The IF statement was following by a logical expression and two statements, the first of which was executed if the logical expression was true. The true and false clauses could be single statements or compound statements delimited by BEGIN and END. A FOR statement was provided for looping through a list which could be either explicitly provided or specified by an arithmetic sequence.

Algol 60 (1960) changed the syntax of the IF statement, separating the conditional, true clause, and false clause with THEN and ELSE instead of parens and semicolons. It also introduced a FOR/WHILE statement. The WHILE clause was a logical expression used to determine whether to perform another iteration of the loop. The FOR/WHILE was not a simple as the WHILE loop of later languages such as Pascal and C, but it had the same expressive power. It would be realized during the 1960s that IF and WHILE statements could be used to replace most occurrences of GOTO statements, and that the resulting code was easier to understand.

Algol 60 added a real (floating point) type for variables. It required that variables have a lexical scope defined by the block containing their declaration. Lexical scope is important for the construction of large programs and it enables recursive functions.

Fortran IV (1962) added the logical operators .AND., .OR., and .NOT. and the comparison operators .GT., .LT., .GE., .LE., .EQ., and .NE. It also added an IF statement which has a logical expression as its conditional.

portability

Fortran I (1956) and Fortran II (1958) contained features specific to the IBM 704. In Fortran IV (1962) these features were removed in the interest of making it possible to write portable code. IBM was interested in porting code to its newer architectures, but customers were interested in porting code to computers manufactured by different companies. Fortran IV was essentially accepted by ANSI as Fortran 66.

numerics

The IBM 704 (1954) was the first mass-produced computer with floating point hardware; IBM sold 123 of them. Floating point implementations were hardware dependent for decades and this was sometimes a portability issue. A floating point standard, IEEE 754, was proposed in 1985. The 80387 (1987) was the first Intel floating point coprocessor to conform to IEEE 754. Starting with the 80486 series of chips (1989) Intel integrated the FPU with the main CPU. Motorola also had an IEEE 754 compliant FPU coprocessor by 1987, the 68881.

IEEE 754 specified a single precision which used 32 bits and a double precision which used 64 bits. For each precision, the specification describes which numbers can be represented exactly, and for those numbers which cannot be represented exactly, how rounding is to be performed. Included among representable values are zero and infinity, both which carry a sign, and "not a number" or NaN.

The specification describes 5 conditions which are indicated by status flags after any arithmetic operation: invalid operation (used for the square root of a negative number), division by zero, overflow, underflow, and inexact.

interpreters and compilers

Interpreters are easier to use and easier to implement, but compilers have a performance advantage and they have been favored by most languages from the introduction of the Fortran I compiler (1956) to well into the 1990s.

Lisp and Basic were two languages from the 1960s that were often implemented with both an interpreter and a compiler. Even in the 1960s it was observed that development went faster with an interpreter. In the 1970s and 1980s a variety of languages appeared which were only interpreted. Some were closely associated with an operating system and used to automate work that could be done at a command prompt. Examples are the languages embedded in Unix shells or the DOS prompt. Others were designed to be embedded in an application. Still others were proposed as glue languages for code written in other languages. The other languages were invoked as separate processes (shell languages) or linked to the interpreter (Tcl). By the mid 1990s, improvement in the speed of hardware and interpreter optimizations made it possible to develop sophisticated applications entirely in an interpreted language such as Perl.

punch cards, terminals, and GUIs

IBM 2260

Teletype terminals such as the ASR-33 (1963) and time-sharing operating systems such as CTSS (1961) made it possible to enter programs into the computer without punch cards and key punches.

Eventually video terminals would be available. One of the first was the IBM 2260 (1964). The IBM 3270 (1972) was popular in mainframe environments. Digital's first video terminal was the VT05 (1970). The VT100 (1978) would introduce escape sequences that would be ANSI standardized and are still used today.

The ASCII character encoding was developed and adopted by 1968.

string and text processing languages

Hollerith constant
The Elliott Algol Input/Output System 1963

Character encoding was initially hardware dependent. The 6-bit BCD encoding was used on the IBM 704 (1954), and the 8-bit EBCDIC was used in System/360 (1964). However, the ASCII encoding, which was mostly finalized by 1963, would become the standard.

Fortran was designed for numerical processing. Character data could be output by concatenating Hollerith constants. It was not until Fortran-77 that the CHARACTER fixed length string data type was added. The FORMAT statement created something akin to the format string passed to *printf* in C. The FORMAT statement could only format numbers in Fortran I, but could include alphanumeric data by Fortran IV (1962).

Algol (1960) was a slight improvement in that it introduced a *string* data type and a string literal format that used left and right leaning single quotes. However, the Algol standard did not specify input and output, which was left to the implementation. It did not specify any string operators, nor did it assume that strings were implemented as arrays.

TODO: Snobol (1962, 1967)

general purpose data structures

operating system access

Web History: The Static Web: 1990

The Original HTTP as defined in 1991
HTML Specification Draft June 1993
WorldWideWeb Browser
Mosaic Web Browser

Tim Berners-Lee created the web in 1990. It ran on a NeXT cube. The browser and the web server communicated via a protocol invented for the purpose called HTTP. The documents were marked up in a type of SGML called HTML. The key innovation was the hyperlink. If the user clicked on a hyperlink, the browser would load the document pointed to by the link. A hyperlink could also take the user to a different section of the current document.

The initial version of HTML included these tags:

```
html, head, title, body, h1, h2, h3, h4, h5, h6, pre, blockquote, b, i, a, img, ul, ol, li, dl, dt, dd
```

The browser developed by Berners-Lee was called WorldWideWeb. It was graphical, but it wasn't widely used because it only ran on NeXT. Nicola Pellow wrote a text-only browser and ported it to a variety of platforms in 1991. Mosaic was developed by Andreesen and others at NCSA and released in February 1993. Mosaic was the first browser which could display images in-line with text. It was originally released for X Windows, and it was ported to Macintosh a few months later. Ports for the Amiga and Windows were available in October and December of 1993.

Web History: CGI and Forms: 1993

RFC 3875: CGI Version 1.1 2004
HTML 2.0 1995
NSAPI Programmer's Guide (pdf) 2000
Apache HTTP Server Project
History of mod_perl
FastCGI Specification 1996

The original web permitted a user to edit a document with a browser, provided he or she had permission to do so. But otherwise the web was static. The group at NCSA developed forms so users could submit data to a web server. They developed the CGI protocol so the server could invoke a separate executable and pass form data to it. The separate executable, referred to as a CGI script in the RFC, could be implemented in almost any language. Perl was a popular early choice. What the CGI script writes to standard out becomes the HTTP response. Usually this would contain a dynamically generated HTML document.

HTML 2.0 introduced the following tags to support forms:

```
form input select option textarea
```

The *input* tag has a *type* attribute which can be one of the following:

```
text password checkbox radio image hidden submit reset
```

If the browser submits the form data with a GET, the form data is included in the URL after a question mark (?). The form data consists of key value pairs. Each key is separated from its value by an equals (=), and the pairs are separated from each other by ampersands (&). The CGI protocol introduces an encoding scheme for escaping the preceding characters in the form data or any other characters that are meaningful or prohibited in URLs. Typically, the web server will set a QUERY_STRING environment variable to pass the GET form data to the CGI script. If the browser submits the data with POST, the form data is encoded in the same manner as for GET, but the data is placed in the HTTP request body. The media type is set to *application/x-www-form-urlencoded*.

Andreesen and others at NCSA joined the newly founded company Netscape, which released a browser in 1994. Netscape also released a web server with a plug-in architecture. The architecture was an attempt to address the fact that handling web requests with CGI scripts was slow: a separate process was created for each request. With the Netscape web server, the equivalent of a CGI script would be written in C and linked in to the server. The C API that the developer used was called NSAPI. Microsoft developed a similar API called ISAPI for the IIS web server.

The NCSA web server had no such plug-in architecture, but it remained the most popular web server in 1995 even though development had come to a halt. The Apache web server project started up that year; it used the NCSA httpd 1.3 code as a starting point and it was the most popular web server within a year. Apache introduced the Apache API, which permitted C style web development in the manner of NSAPI and ISAPI. The Apache extension *mod_perl*, released in March 1996, was a client of the Apache API. By means of *mod_perl* an Apache web server could handle a CGI request in memory using an embedded perl interpreter instead of forking off a separate perl process.

TODO: cookies
TODO: tables

Web History: HTML Templates: 1995

PHP/FI Version 2.0
PHP Usage

Web development with CGI scripts written in Perl was easier than writing web server plug-ins in C. The task of writing Perl CGI scripts was made easier by libraries such as cgi-lib.pl and CGI.pm. These libraries made the query parameters available in a uniform fashion regardless of whether a GET or POST request was being handled and also took care of assembling the headers in the response. Still, CGI scripts tended to be difficult to maintain because of the piecemeal manner in which the response document is assembled.

Rasmus Lerdorf adopted a template approach for maintaining his personal home page. The document to be served up was mostly static HTML with an escaping mechanism for inserting snippets of code. In version 2.0 the escapes were <? *code* > and <?echo *code* >. Lerdorf released the code for the original version, called PHP/FI and implemented in Perl, in 1995. The original version was re-implemented in C and version 2.0 was released in 1997. For version 3.0, released in 1998, the name was simplified to PHP. Versions 4.0 and 5.0 were released in 2000 and 2004. PHP greatly increased in popularity with the release of version 4.0. Forum software, blogging software, wikis, and other content management systems (CMS) are often implemented in PHP.

Microsoft added a tempate engine called Active Server Pages (ASP) for IIS in 1996. ASP uses <% *code* %> and <%= *code* %> for escapes; the code inside the script could be any number of languages but was usually a dialect of Visual Basic called VBScript. Java Server Pages (JSP), introduced by Sun

in 1999, uses the same escapes to embed Java.

Web History: MVC Frameworks: 2000

The template approach to web development has limitations. Consider the case where the web designer wants to present a certain page if the user is logged in, and a completely unrelated page if the user is not logged in. If the request is routed to an HTML template, then the template will likely have to contain a branch and two mostly unrelated HTML templates. The page that is presented when the user is not logged in might also be displayed under other circumstances, and unless some code sharing mechanism is devised, there will be duplicate code and the maintenance problem that entails.

The solution is for the request to initially be handled by a controller. Based upon the circumstances of the request, the controller chooses the correct HTML template, or view, to present to the user.

Websites frequently retrieve data from and persist data to a database. In a simple PHP website, the SQL might be placed directly in the HTML template. However, this results in a file which mixes three languages: SQL, HTML, and PHP. It is cleaner to put all database access into a separate file or model, and this also promotes code reuse.

The Model-View-Controller design pattern was conceived in 1978. It was used in Smalltalk for GUI design. It was perhaps in Java that the MVC pattern was introduced to web development.

Early versions of Java were more likely to be run in the browser as an applet than in the server. Sun finalized the Servlet API in June 1997. Servlets handled requests and returned responses, and thus were the equivalent of controllers in the MVC pattern. Sun worked on a reference web server which used servlets. This code was donated to the Apache foundation, which used it in the Tomcat webserver, released in 1999. The same year Sun introduced JSP, which corresponds to the view of the MVC pattern.

The Struts MVC framework was introduced in 2000. The Spring MVC framework was introduced in 2002; some prefer it to Struts because it doesn't use Enterprise JavaBeans. Hibernate, introduced in 2002, is an ORM and can serve as the model of an MVC framework.

Ruby on Rails was released in 2004. Ruby has a couple of advantages over Java when implementing an MVC framework. The models can inspect the database and create accessor methods for each column in the underlying table on the fly. Ruby is more concise than Java and has better string manipulation features, so it is a better language to use in HTML templates. Other dynamic languages have built MVC frameworks, e.g. Django for Python.

TODO: javascript vs java
TODO: css
TODO: ajax
TODO: flash and html5

Perl Version History

From the Perl Timeline and CPAN README:

Perl 1.0 (gzipped tar) Dec 18, 1987

Perl 2.0 (gizpped tar) Jun 5, 1988

- New regexp routines derived from Henry Spencer's.
  - Support for /(foo|bar)/.
  - Support for /(foo)*/ and /(foo)+/.
  - \s for whitespace, \S for non-, \d for digit, \D nondigit
- Local variables in blocks, subroutines and evals.
- Recursive subroutine calls are now supported.
- Array values may now be interpolated into lists: unlink 'foo', 'bar', @trashcan, 'tmp';
- File globbing.
- Use of <> in array contexts returns the whole file or glob list.
- New iterator for normal arrays, foreach, that allows both read and write.
- Ability to open pipe to a forked off script for secure pipes in setuid scripts.
- File inclusion via do 'foo.pl';
- More file tests, including -t to see if, for instance, stdin is a terminal. File tests now behave in a more correct manner. You can do file tests on filehandles as well as filenames. The special filetests -T and -B test a file to see if it's text or binary.
- An eof can now be used on each file of the <> input for such purposes as resetting the line numbers or appending to each file of an inplace edit.
- Assignments can now function as lvalues, so you can say things like ($HOST = $host) =~ tr/a-z/A-Z/; ($obj = $src) =~ s/\.c$/.o/;
- You can now do certain file operations with a variable which holds the name of a filehandle, e.g. open(++$incl,$includefilename); $foo = <$incl>;
- Warnings are now available (with -w) on use of uninitialized variables and on identifiers that are mentioned only once, and on reference to various undefined things.
- There is now a wait operator.
- There is now a sort operator.
- The manual is now not lying when it says that perl is generally faster than sed. I hope.

Perl 3.0 (gzipped tar) Oct 18, 1989)

- Perl can now handle binary data correctly and has functions to pack and unpack binary structures into arrays or lists. You can now do arbitrary ioctl functions.
- You can now pass things to subroutines by reference.
- Debugger enhancements.
- An array or associative array may now appear in a local() list.
- Array values may now be interpolated into strings.
- Subroutine names are now distinguished by prefixing with &. You can call subroutines without using do, and without passing any argument list at all.
- You can use the new -u switch to cause perl to dump core so that you can run undump and produce a binary executable image. Alternately you can use the "dump" operator after initializing any variables and such.
- You can now chop lists.
- Perl now uses /bin/csh to do filename globbing, if available. This means that filenames with spaces or other strangenesses work right.
- New functions: mkdir and rmdir, getppid, getpgrp and setpgrp, getpriority and setpriority, chroot, ioctl and fcntl, flock, readlink, lstat, rindex, pack and unpack, read, warn, dbmopen and dbmclose, dump, reverse, defined, undef.

Perl 4.0 (gzipped tar) Mar 21, 1991

- According to wikipedia, this was not a major change. The version was bumped solely because of the publication of the camel book.

Perl 5.0 (Oct 18, 1994)

- Objects.
- The documentation is much more extensive and perldoc along with pod is introduced.
- Lexical scoping available via my. eval can see the current lexical variables.
- The preferred package delimiter is now :: rather than '.
- New functions include: abs(), chr(), uc(), ucfirst(), lc(), lcfirst(), chomp(), glob()
- There is now an English module that provides human readable translations for cryptic variable names.
- Several previously added features have been subsumed under the new keywords use and no.
- Pattern matches may now be followed by an m or s modifier to explicitly request multiline or singleline semantics. An s modifier makes . match newline.
- @ now always interpolates an array in double-quotish strings. Some programs may now need to use backslash to protect any @ that shouldn't interpolate.
- It is no longer syntactically legal to use whitespace as the name of a variable, or as a delimiter for any kind of quote construct.
- The -w switch is much more informative.
- => is now a synonym for comma. This is useful as documentation for arguments that come in pairs, such as initializers for associative arrays, or named arguments to a subroutine.

Perl 5.4 (aka 5.004) (May 1997)

Perl 5.5 (aka 5.005) (July 1998)

5.6 Mar 28, 2000

- Several experimental features, including: support for Unicode, fork() emulation on Windows, 64-bit support, lvalue subroutines, weak references, and new regular expression constructs. See below for the full list.
- Standard internal representation for strings is UTF-8. (EBCDIC support has been discontinued because of this.)
- Better support for interpreter concurrency.
- Lexically scoped warning categories.
- "our" declarations for global variables.
- String literals can be written using character ordinals. For example, v102.111.111 is the same as "foo".
- New syntax for subroutine attributes. (The attrs pragma is now deprecated.)
- Filehandles can be autovivified. For example: open my $foo, $file or die;
- open() may be called with three arguments to avoid magic behavior.
- Support for large files, where available (will be enabled by default.)
- CHECK blocks. These are like END blocks, but will be called when the compilation of the main program ends.
- POSIX character class syntax supported, e.g. /[[:alpha:]]/
- pack() and unpack() support null-terminated strings, native data types, counted strings, and comments in templates
- Support for binary numbers.
- exists() and delete() work on array elements. Existence of a subroutine (as opposed to its defined-ness) may also be checked with exists(&sub)).
- Where possible, Perl does the sane thing to deal with buffered data automatically.
- binmode() can be used to set :crlf and :raw modes on dosish platforms. The open pragma does the same in the lexical scope, allowing the mode to be set for backticks.
- Many modules now come standard, including Devel::DProf, Devel::Peek, and Pod::Parser.
- Many modules have been substantially revised or rewritten.

- The JPL ("Java Perl Lingo") distribution comes bundled with Perl.
- Most platform ports have improved functionality. Support for EBCDIC platforms has been withdrawn due to standardization on UTF-8.
- Much new documentation in the form of tutorials and reference information has been added.
- Plenty of bug fixes.

Perl 5.8 (July 2002)

Perl 5.10 (Dec 2007)

Perl 5.12 (Apr 2010)

- Unicode implemented according to Unicode standard
- improvements to time, fix to Y2038
- version numbers in package statements

What's planned for Perl 6

Python Version History

Python History Blog by Guido van Rossum
Brief Timeline of Python:
History of Python

0.9 (Feb 20, 1991)

- classes with inheritance
- exception handling
- list, dict, str datatypes
- modules

1.0 (Jan 26, 1994)

- lambda, map, filter, reduce

1.1 (Oct 11, 1994)

1.2 (Apr 13, 1995)

1.3 (Oct 13, 1995)

1.4 (Oct 25, 1996)

1.5 (Jan 3, 1998)

- exceptions are classes, not strings
- re module (perl style regular expressions) replaces regex

- nested modules (i.e. hierarchical namespace)

2.0 (Oct 16, 2000)

- unicode
- list comprehensions
- augmented assignment (+=, *=, etc.)
- cycle detection added to garbage collector

2.1 (Apr 17, 2001)

- nested lexical scope
- classes able to override all comparison operators individually

2.2 (Dec 21, 2001)

- introduction of new-style classes
- descriptors and properties
- ability to subclass built-in classes
- class and static methods
- callbacks for object property access
- ability to restrict settable attributes to class defined set
- base class object added for all built-ins
- for generalized from sequences to all objects with iter()
- integers and long integers unified, removing some overflow errors
- // is integer division, and / is now float division
- add generators and the yield statement

2.3 (Jul 29, 2003)

- sets module
- generators/yield no longer optional
- source code encodings (for string literals only): # -*- coding: UTF-8 -*-
- boolean type added

2.4 (Nov 30, 2004)

- sets built-in
- Template class for string substitutions (in addition to older % operator)
- generator expressions (when list comprehensions use too much memory)
- decorator functions for type checking
- decimal data type with user specified precision

2.5 (Sep 16, 2006)

- conditional expressions

- partial function evaluation: partial()
- unifed try/except/finally
- with statement and context management protocol (must be imported from __future__)

2.6 (Oct 1, 2008)

- -3 command line switch warns about use of features absent in python 3.0
- with statement does not require import
- multiprocessing package (processes communicate via queues, threadlike syntax for process management)
- str.format()
- as keyword in except clause: except TypeError as exc (old syntax still supported: except TypeError, exc)
- Abstract Base Classes (ABC): Container, Iterable; user can define an ABC with ABCMeta
- binary string literal b'', binary and octal numbers 0b01010 and 0o1723
- class decorators
- number hierarchy expansion: Number, Fraction, Integral, Complex

3.0 (Dec 3, 2008)

- old style classes no longer available
- print is function instead of statement (parens around argument mandatory)
- comma in except clause discontinued: (except TypeError, exc)
- 1/2 returns float instead of int (1//2, available since 2.2, returns int)
- strings (str) are always Unicode; bytes data type available for arbitrary 8-bit data

3.1 (Jun 27, 2009)

- ordered dictionaries
- format specifier for thousands separator

2.7 (Jul 3, 2010)

- adds ordered dictionaries and format specifier for thousands separator from 3.1 to 2.X

Ruby Version History

0.95 (Dec 21, 1995)

1.0 (Dec 25, 1996)

1.4.0 (Aug 13, 1999)

1.6.0 (Sep 19, 2000)

Pickaxe Book (Dec 15, 2001)

1.8.0 (Aug 4, 2003)

1.8.1 (Dec 25, 2003)

Rails (Jul 2004)

1.8.2 (Dec 25, 2004)

1.8.3 (Sep 21, 2005)

1.8.4 (Dec 24, 2005)

1.8.5 (Aug 25, 2006)

1.8.6 (Mar 13, 2007)

1.8.7 (May 31, 2008)

1.9.1 (Jan 31, 2009)