Perl Beginners' Site

Perl - because programming should be fun.

Hyperpolyglot - Sheet 1

Learn Perl Now!
And get a job doing Perl.
Interpreted Languages: PHP, Perl, Python, Ruby (Sheet One)

a side-by-side reference sheet

	php	perl	python	
versions used	*5.3*	*5.12; 5.14*	*2.7; 3.2*	*1.8; 1.9*
implicit prologue	*none*	`use strict;`	`import os, re, sys`	*none*
show version	`$ php --version`	`$ perl --version`	`$ python -V`	`$ ruby --`

grammar and invocation

	php	perl	python	
interpreter	`$ php -f foo.php`	`$ perl foo.pl`	`$ python foo.py`	`$ ruby fo`
repl	`$ php`	`$ perl -de 0`	`$ python`	`$ irb`
command line program	`$ php -r 'echo "hi\n";'`	`$ perl -e 'print("hi\n")'`	`$ python -c "print('hi')"`	`$ ruby -e`
block delimiters	`{}`	`{}`	*: and offside rule*	`{}` `do end`
statement separator	`;` *statements must be semicolon terminated inside {}*	`;`	*newline or ;* *newlines not separators inside (), [], {}, triple quote literals, or after backslash:* `\`	*newline o* *newlines* *`, ''', "* *backslash*
end-of-line comment	`// comment` `# comment`	`# comment`	`# comment`	`# comment`

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

variables and expressions

	php	perl	python	
local variable	# in function body: $v = NULL; $a = array(); $d = array(); $x = 1; list($y, $z) = array(2, 3);	my $v; my (@a, %d); my $x = 1; my ($y, $z) = (2, 3);	# in function body: v = None a, d = [], {} x = 1 y, z = 2, 3	v = nil a, d = [] x = 1 y, z = 2,
regions which define lexical scope	*top level:* *function or method body* *nestable (with use clause):* *anonymous function body*	*top level:* *file* *nestable:* *function body* *anonymous function body* *anonymous block*	*nestable (read only):* *function or method body*	*top level* *file* *class b* *module* *method* *nestable:* *anonymo* *anonymo*
	list($g1, $g2) = array(7, 8);	our ($g1, $g2) = (7, 8);	g1, g2 = 7, 8	$g1, $g2
global variable	function swap_globals() { global $g1, $g2; list($g1, $g2) = array($g2, $g1); }	sub swap_globals { ($g1, $g2) = ($g2, $g1); }	def swap_globals(): global g1, g2 g1, g2 = g2, g1	def swap_ $g1, $g end
constant	define("PI", 3.14);	use constant PI => 3.14;	# uppercase identifiers # constant by convention PI = 3.14	# warning # identif PI = 3.14
assignment	$v = 1;	$v = 1;	*assignments can be chained but otherwise don't return values:* v = 1	v = 1
	list($x, $y, $z) = array(1 ,2, 3);	($x, $y, $z) = (1, 2, 3);	x, y, z = 1, 2, 3	x, y, z =
parallel assignment	# 3 is discarded: list($x, $y) = array(1, 2, 3);	# 3 is discarded: ($x, $y) = (1, 2, 3);	# raises ValueError: x, y = 1, 2, 3	# 3 is di x, y = 1,
	# $z set to NULL: list($x, $y, $z) = array(1, 2);	# $z set to undef: ($x, $y, $z) = (1, 2);	# raises ValueError: x, y, z = 1, 2	# z set t x, y, z =
swap	list($x, $y) = array($y, $x);	($x, $y) = ($y, $x);	x, y = y, x	x, y = y,
compound assignment *arithmetic, string, logical, bit*	+= -= *= *none* /= %= **= .= *none* &= |= *none* <<= >>= &= |= ^=	+= -= *= *none* /= %= **= .= x= &&= ||= ^= <<= >>= &= |= ^=	# do not return values: += -= *= /= //= %= **= += *= &= |= ^= <<= >>= &= |= ^=	+= -= *= += *= &&= ||= ^ <<= >>= &
increment and decrement	$x = 1; $y = ++$x; $z = --$y;	my $x = 1; my $y = ++$x; my $z = --$y;	*none*	x = 1 # x and y y = x.suc z = y.pre
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* Na
conditional expression	$x > 0 ? $x : -$x	$x > 0 ? $x : -$x	x if x > 0 else -x	x > 0 ? x

arithmetic and logic

	php	perl	python	
true and false	TRUE FALSE # case insensitve	1 ""	True False	true fals
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	&& || ! *lower precedence:* and or xor not	and or not	&& || ! *lower pre* and or no
relational operators	== != *or* <> > < >= <= *no conversion:* === !==	*numbers only:* == != > < >= <= *strings:* eq ne gt lt ge le	*relational operators are chainable:* == != > < >= <=	== != > <

		use List::Util qw(min max);		
min and max	min(1,2,3) max(1,2,3) $a = array(1,2,3) min($a) max($a)	min(1,2,3); max(1,2,3); @a = (1,2,3); min(@a); max(@a);	min(1,2,3) max(1,2,3) min([1,2,3]) max([1,2,3])	[1,2,3].m [1,2,3].m
three value comparison	*none*	0 <=> 1 "do" cmp "re"	*removed from Python 3:* cmp(0, 1) cmp('do', 're')	0 <=> 1 "do" <=>
arithmetic operators *addition, subtraction, multiplication, float division, quotient, remainder*	+ - * / *none* %	+ - * / *none* %	+ - * *see note* // % *Python 2 does not have an operator which performs float division on integers. In Python 3 / always performs float division.*	+ - * x.f
integer division	(int) (13 / 5)	int (13 / 5)	13 // 5	13 / 5
divmod	*none*	*none*	q, r = divmod(13, 5)	q, r = 13
integer division by zero	*returns* FALSE *with warning*	*error*	*raises* ZeroDivisionError	*raises* Ze
float division	13 / 5	13 / 5	float(13) / 5 # Python 3: 13 / 5	13.to_f / 13.fdiv(5
float division by zero	*returns* FALSE *with warning*	*error*	*raises* ZeroDivisionError	*returns* -
power	pow(2, 32)	2**32	2**32	2**32
sqrt	sqrt(2)	sqrt(2)	import math math.sqrt(2)	include M sqrt(2)
sqrt -1	NaN	*error unless* use Math::Complex *in effect*	# raises ValueError: import math math.sqrt(-1) # returns complex float: import cmath cmath.sqrt(-1)	*raises* Er
transcendental functions	exp log sin cos tan asin acos atan atan2	use Math::Trig qw(tan asin acos atan); exp log sin cos tan asin acos atan atan2	from math import exp, log, \ sin, cos, tan, asin, acos, atan, atan2	include M exp log s
transcendental constants *π and e*	M_PI M_E		import math math.pi math.e	include M PI E
float truncation	(int)$x round($x) ceil($x) floor($x)	# cpan -i Number::Format use Number::Format 'round'; use POSIX qw(ceil floor); int($x) round($x, 0) ceil($x) floor($x)	import math int(x) int(round(x)) math.ceil(x) math.floor(x)	x.to_i x.round x.ceil x.floor
absolute value	abs($x)	abs($x)	abs(x)	x.abs
integer overflow	*converted to float*	*converted to float; use* Math::BigInt *to create arbitrary length integers*	*becomes arbitrary length integer of type* long	*becomes a* Bignum
float overflow	INF	inf	*raises* OverflowError	Infinity
rational construction	*none*	use Math::BigRat; my $x = Math::BigRat->new("22/7");	from fractions import Fraction x = Fraction(22,7)	require ' x = Ratio
rational decomposition	*none*	$x->numerator(); $x->denominator();	x.numerator x.denominator	x.numerat x.denomin

complex construction	*none*	`use Math::Complex;` `my $z = 1 + 1.414 * i;`	`z = 1 + 1.414j`	`require '` `z = 1 + 1`
complex decomposition *real and imaginary component, argument, absolute value, conjugate*	*none*	`Re($z);` `Im($z);` `arg($z);` `abs($z);` `~$z;`	`import cmath` `z.real` `z.imag` `cmath.phase(z)` `abs(z)` `z.conjugate()`	`z.real` `z.imag` `z.arg` `z.abs` `z.conj`
random number *uniform integer, uniform float, normal float*	`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*
random seed *set, get, restore*	`srand(17);` *none*	`srand 17;` `my $seed = srand;` `srand($seed);`	`import random` `random.seed(17)` `seed = random.getstate()` `random.setstate(seed)`	`srand(17)` `seed = sr` `srand(see`
bit operators	`<< >> & | ^ ~`	`<< >> & | ^ ~`	`<< >> & | ^ ~`	`<< >> & |`
binary, octal, and hex literals	*none* `052` `0x2a`	`0b101010` `052` `0x2a`	`0b101010` `052` `0x2a`	`0b101010` `052` `0x2a`
base conversion	`base_convert("42", 10, 7);` `base_convert("60", 7, 10);`	`# cpan -i Math::BaseCalc` `use Math::BaseCalc;` `$c = new Math::BaseCalc(digits=> [0..6]);` `$c->to_base(42);` `$c->from_base("60");`	*none* `int("60", 7)`	`42.to_s(7` `"60".to_i`

strings

	php	perl	python	
string literal	`"don't say \"no\""` `'don\'t say "no"'`	`"don't say \"no\""` `'don\'t say "no"'`	`'don\'t say "no"'` `"don't say \"no\""` `"don't " 'say "no"'` `'''don't say "no"'''` `"""don't say "no\""""`	`"don't sa` `'don\'t s` `"don't "`
newline in literal	*yes*	*yes*	*triple quote literals only*	*yes*
character escapes	*double quoted:* `\f \n \r \t \v \xhh \$ \" \\ \ooo` *single quoted:* `\' \\`	*double quoted:* `\a \b \cx \e \f \n \r \t \xhh \x{hhhh}` `\ooo` *Perl 5.14:* `\o{ooo}` *single quoted:* `\' \\`	*single and double quoted:* `\newline \\ \' \" \a \b \f \n \r \t \v` `\ooo \xhh` *Python 3:* `\uhhhh \Uhhhhhhhh`	*double qu* `\a \b \cx` *Ruby 1.* *single qu* `\' \\`
variable interpolation	`$count = 3;` `$item = "ball";` `echo "$count ${item}s\n";`	`my $count = 3;` `my $item = "ball";` `print "$count ${item}s\n";`	`count = 3` `item = 'ball'` `print('{count} {item}s'.format(` `  **locals()))`	`count = 3` `item = "b` `puts "#{c`
custom delimiters	*none*	`my $s1 = q(lorem ipsum);` `my $s2 = qq($s1 dolor sit amet);`	*none*	`s1 = %q(l` `s2 = %Q(#`
sprintf	`$fmt = "lorem %s %d %f";` `sprintf($fmt, "ipsum", 13, 3.7);`	`my $fmt = "lorem %s %d %f";` `sprintf($fmt, "ipsum", 13, 3.7)`	`'lorem %s %d %f' % ('ipsum', 13, 3.7)` `fmt = 'lorem {0} {1} {2}'` `fmt.format('ipsum', 13, 3.7)`	`"lorem %s`
	`$word = "amet";`	`$word = "amet";`		`word = "a`
here document	`$s = <<<EOF` `lorem ipsum` `dolor sit $word` `EOF;`	`$s = <<EOF;` `lorem ipsum` `dolor sit $word` `EOF`	*none*	`s = <<EOF` `lorem ips` `dolor sit` `EOF`
concatenate	`$s = "Hello, ";` `$s2 = $s . "World!";`	`my $s = "Hello, ";` `my $s2 = $s . "World!";`	`s = 'Hello, '` `s2 = s + 'World!'` *juxtaposition can be used to concatenate literals:* `s2 = 'Hello, ' "World!"`	`s = "Hell` `s2 = s +` *juxtaposi literals:* `s2 ="Hell`
replicate	`$hbar = str_repeat("-", 80);`	`my $hbar = "-" x 80;`	`hbar = '-' * 80`	`hbar = "-`

split *simple split, split in two, keep delimiters, split into chars*	explode(" ", "do re mi fa") preg_split('/\s+/', "do re mi fa", 2) preg_split('/(\s+)/', "do re mi fa", NULL, PREG_SPLIT_DELIM_CAPTURE); str_split("abcd")	split(/\s+/, "do re mi fa") split(/\s+/, "do re mi fa", 2) split/(\s+)/, "do re mi fa"); split//, "abcd")	'do re mi fa'.split() 'do re mi fa'.split(None, 1) re.split('(\s+)', 'do re mi fa') list('abcd')	"do re mi "do re mi "do re mi "abcd".sp
join	$a = array("do", "re", "mi", "fa"); implode(" ", $a)	join(" ", qw(do re mi fa))	' '.join(['do', 're', 'mi', 'fa'])	%w(do re
case manipulation	strtoupper("lorem") strtolower("LOREM") ucfirst("lorem")	uc("lorem") lc("LOREM") ucfirst("lorem")	'lorem'.upper() 'LOREM'.lower() 'lorem'.capitalize()	"lorem".u "LOREM".d "lorem".c
trim *both sides, left, right*	trim(" lorem ") ltrim(" lorem") rtrim("lorem ")	# cpan -i Text::Trim use Text::Trim; trim " lorem " ltrim " lorem" rtrim "lorem " # cpan -i Text::Trim use Text::Format;	' lorem '.strip() ' lorem'.lstrip() 'lorem '.rstrip()	" lorem " " lorem". "lorem ".
pad *on right, on left, centered*	str_pad("lorem", 10) str_pad("lorem", 10, " ", STR_PAD_LEFT) str_pad("lorem", 10, " ", STR_PAD_BOTH)	sprintf("%-10s", "lorem") sprintf("%10s", "lorem") $text = Text::Format->new(columns => 10); $text->center("lorem");	'lorem'.ljust(10) 'lorem'.rjust(10) 'lorem'.center(10)	"lorem".l "lorem".r "lorem".c
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". 73.9 + ". "value: "
length	strlen("lorem")	length("lorem")	len('lorem')	"lorem".l "lorem".s
index of substring	strpos("do re re", "re") strrpos("do re re", "re") *return* FALSE *if not found*	index("lorem ipsum", "ipsum") rindex("do re re", "re") *return* -1 *if not found*	'do re re'.index('re') 'do re re'.rindex('re') *raise* ValueError *if not found*	"do re re "do re re *return* ni
extract substring	substr("lorem ipsum", 6, 5)	substr("lorem ipsum", 6, 5)	'lorem ipsum'[6:11]	"lorem ip
extract character	*syntax error to use index notation directly on string literal:* $s = "lorem ipsum"; $s[6];	*can't use index notation with strings:* substr("lorem ipsum", 6, 1)	'lorem ipsum'[6]	"lorem ip
chr and ord	chr(65) ord("A")	chr(65) ord("A")	chr(65) ord('A')	65.chr "A"[0]
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".t

regular expresions

	php	perl	python	
literal, custom delimited literal	'/lorem|ipsum/' '(/etc/hosts)'	/lorem|ipsum/ qr(/etc/hosts)	re.compile('lorem|ipsum') *none*	/lorem|ip %r(/etc/h
character class abbreviations and anchors	*char class abbrevs:* . \d \D \h \H \s \S \v \V \w \W *anchors:* ^ $ \A \b \B \z \Z	*char class abbrevs:* . \d \D \h \H \s \S \v \V \w \W *anchors:* ^ $ \A \b \B \z \Z	*char class abbrevs:* . \d \D \s \S \w \W *anchors:* ^ $ \A \b \B \Z	*char clas* . \d \D \ *anchors:*
match test	if (preg_match('/1999/', $s)) { echo "party!\n"; }	if ($s =~ /1999/) { print "party!\n"; }	if re.search('1999', s): print('party!')	if /1999/ puts "p end
case insensitive match test	preg_match('/lorem/i', "Lorem")	"Lorem" =~ /lorem/i	re.search('lorem', 'Lorem', re.I)	/lorem/i.r
modifiers	e i m s x	i m s p x	re.I re.M re.S re.X	i o m x
substitution	$s = "do re mi mi mi"; $s = preg_replace('/mi/', "ma", $s);	my $s = "do re mi mi mi"; $s =~ s/mi/ma/g;	s = 'do re mi mi mi' s = re.compile('mi').sub('ma', s)	s = "do r s.gsub!(/
match, prematch, postmatch	*none*	if ($s =~ /\d{4}/p) { $match = ${^MATCH}; $prematch = ${^PREMATCH}; $postmatch = ${^POSTMATCH}; }	m = re.search('\d{4}', s) if m: match = m.group() prematch = s[0:m.start(0)] postmatch = s[m.end(0):len(s)]	m = /\d{4 if m match = prematc postmat end

	php	perl	python	
group capture	$s = "2010-06-03"; $rx = '/(\d{4})-(\d{2})-(\d{2})/'; preg_match($rx, $s, $m); list($_, $yr, $mo, $dy) = $m;	$rx = qr/(\d{4})-(\d{2})-(\d{2})/; "2010-06-03" =~ $rx; ($yr, $mo, $dy) = ($1, $2, $3);	rx = '(\d{4})-(\d{2})-(\d{2})' m = re.search(rx, '2010-06-03') yr, mo, dy = m.groups()	rx = /(\d m = rx.ma yr, mo, d
named group capture			rx = '^(?P<file>.+)\.(?P<suffix>.+)$' m = re.search(rx, 'foo.txt') m.groupdict()['file'] m.groupdict()['suffix']	
scan	$s = "dolor sit amet"; preg_match_all('/\w+/', $s, $m); $a = $m[0];	my $s = "dolor sit amet"; @a = $s =~ m/\w+/g;	s = 'dolor sit amet' a = re.findall('\w+', s)	a = "dolo
backreference in match and substitution	preg_match('/(\w+) \1/', "do do") $s = "do re"; $rx = '/(\w+) (\w+)/'; $s = preg_replace($rx, '\2 \1', $s);	"do do" =~ /(\w+) \1/ my $s = "do re"; $s =~ s/(\w+) (\w+)/$2 $1/;	*none* rx = re.compile('(\w+) (\w+)') rx.sub(r'\2 \1', 'do re')	/(\w+) \1 "do re".s
recursive regex	'/\(([^()]*|($R))\)/'	/\(([^()]*|(?R))\)/	*none*	*Ruby 1.9:* /(?<p>\((

dates and time

	php	perl	python	
date/time type	DateTime	Time::Piece *if* use Time::Piece *in effect, otherwise tm array*	datetime.datetime	Time
current date/time	$t = new DateTime("now"); $utc_tmz = new DateTimeZone("UTC"); $utc = new DateTime("now", $utc_tmz);	use Time::Piece; my $t = localtime(time); my $utc = gmtime(time);	import datetime t = datetime.datetime.now() utc = datetime.datetime.utcnow()	t = Time. utc = Tim
to unix epoch, from unix epoch	$epoch = $t->getTimestamp(); $t2 = new DateTime(); $t2->setTimestamp(1304442000);	use Time::Local; use Time::Piece; my $epoch = timelocal($t); my $t2 = localtime(1304442000);	from datetime import datetime as dt epoch = int(t.strftime("%s")) t2 = dt.fromtimestamp(1304442000)	epoch = t t2 = Time
current unix epoch	$epoch = time();	$epoch = time;	import datetime t = datetime.datetime.now() epoch = int(t.strftime("%s"))	epoch = T
strftime	strftime("%Y-%m-%d %H:%M:%S", $epoch); date("Y-m-d H:i:s", $epoch); $t->format("Y-m-d H:i:s");	use Time::Piece; $t = localtime(time); $fmt = "%Y-%m-%d %H:%M:%S"; print $t->strftime($fmt);	t.strftime('%Y-%m-%d %H:%M:%S')	t.strftim
default format example	*no default string representation*	Tue Aug 23 19:35:19 2011	2011-08-23 19:35:59.411135	2011-08-2
strptime	$fmt = "Y-m-d H:i:s"; $s = "2011-05-03 10:00:00"; $t = DateTime::createFromFormat($fmt, $s);	use Time::Local; use Time::Piece; $s = "2011-05-03 10:00:00"; $fmt = "%Y-%m-%d %H:%M:%S"; $t = Time::Piece->strptime($s,$fmt);	from datetime import datetime s = '2011-05-03 10:00:00' fmt = '%Y-%m-%d %H:%M:%S' t = datetime.strptime(s, fmt)	require ' s = "2011 fmt = "%Y t = Date.
parse date w/o format	$epoch = strtotime("July 7, 1999");	# cpan -i Date::Parse use Date::Parse; $epoch = str2time("July 7, 1999");	# pip install python-dateutil import dateutil.parser s = 'July 7, 1999' t = dateutil.parser.parse(s)	require ' s = "July t = Date.
result of date subtraction	DateInterval *object if* diff *method used:* $fmt = "Y-m-d H:i:s"; $s = "2011-05-03 10:00:00"; $then = DateTime::createFromFormat($fmt, $s); $now = new DateTime("now"); $interval = $now->diff($then);	Time::Seconds *object if* use Time::Piece *in effect; not meaningful to subtract tm arrays*	datetime.timedelta *object*	Float *con seconds*
add time duration	$now = new DateTime("now"); $now->add(new DateInterval("PT10M3S");	use Time::Seconds; $now = localtime(time); $now += 10 * ONE_MINUTE() + 3;	import datetime delta = datetime.timedelta(minutes=10, seconds=3) t = datetime.datetime.now() + delta	require ' s = "10 m delta = D t = Time.
local timezone	*DateTime objects can be instantiated without specifying the timezone if a default is set:* $s = "America/Los_Angeles"; date_default_timezone_set($s);	Time::Piece *has local timezone if created with* localtime *and UTC timezone if created with* gmtime; *tm arrays have no timezone or offset info*	*a* datetime *object has no timezone information unless a* tzinfo *object is provided when it is created*	*if no tim timezone*
arbitrary timezone			# pip install pytz import pytz import datetime	# gem ins require '

			tmz = pytz.timezone('Asia/Tokyo') utc = datetime.datetime.utcnow() utc_dt = datetime.datetime(*utc.timetuple()[0:5], tzinfo=pytz.utc) jp_dt = utc_dt.astimezone(tmz)	tmz = TZI jp_time =
timezone name; offset from UTC; is daylight savings?	$tmz = date_timezone_get($t); timezone_name_get($tmz); date_offset_get($t) / 3600; $t->format("I");	# cpan -i DateTime use DateTime; use DateTime::TimeZone; $dt = DateTime->now(); $tz = DateTime::TimeZone->new(name=>"local"); $tz->name; $tz->offset_for_datetime($dt) / 3600; $tz->is_dst_for_datetime($dt);	import time tm = time.localtime() time.tzname[tm.tm_isdst] (time.timezone / -3600) + tm.tm_isdst tm.tm_isdst	t.zone t.utc_off t.dst?
microseconds	list($frac, $sec) = explode(" ", microtime()); $usec = $frac * 1000 * 1000;	use Time::HiRes qw(gettimeofday); ($sec, $usec) = gettimeofday;	t.microsecond	t.usec
sleep	*a float argument will be truncated to an integer:* sleep(1);	*a float argument will be truncated to an integer:* sleep 1;	import time time.sleep(0.5)	sleep(0.5
timeout	*use* set_time_limit *to limit execution time of the entire script; use* stream_set_timeout *to limit time spent reading from a stream opened with* fopen *or* fsockopen	eval { $SIG{ALRM}= sub {die "timeout!";}; alarm 5; sleep 10; }; alarm 0;	import signal, time class Timeout(Exception): pass def timeout_handler(signo, fm): raise Timeout() signal.signal(signal.SIGALRM, timeout_handler) try: signal.alarm(5) time.sleep(10) except Timeout: pass signal.alarm(0)	require ' begin Timeout sleep end rescue Ti end

arrays

	php	perl	python	
literal	$a = array(1, 2, 3, 4);	@a = (1, 2, 3, 4);	a = [1, 2, 3, 4]	a = [1, 2
quote words	*none*	@a = qw(do re mi);	*none*	a = %w(do
size	count($a)	$#a + 1 *or* scalar(@a)	len(a)	a.size a.length
empty test	!$a	!@a	not a	NoMethodE a.empty?
lookup	$a[0]	$a[0]	a[0]	a[0]
update	$a[0] = "lorem";	$a[0] = "lorem";	a[0] = 'lorem'	a[0] = "l
out-of-bounds behavior	$a = array(); *evaluates as* NULL: $a[10]; *increases array size to one:* $a[10] = "lorem";	@a = (); *evaluates as* undef: $a[10]; *increases array size to 11:* $a[10] = "lorem";	a = [] *raises* IndexError: a[10] *raises* IndexError: a[10] = 'lorem'	a = [] *evaluates* a[10] *increases* a[10] = "
index of array element	$a = array("x", "y", "z", "w"); $i = array_search("y", $a);	use List::Util 'first'; @a = qw(x y z w); $i = first {$a[$_] eq "y"} (0..$#a);	a = ['x', 'y', 'z', 'w'] i = a.index('y')	a = %w(x i = a.ind
slice by endpoints, by length	*select 3rd and 4th elements:* *none* array_slice($a, 2, 2)	*select 3rd and 4th elements:* @a[2..3] splice(@a, 2, 2)	*select 3rd and 4th elements:* a[2:4] a[2:2 + 2]	*select 3r* a[2..3] a[2, 2]
slice to end	array_slice($a, 1)	@a[1..$#a]	a[1:]	a[1..-1]
manipulate back	$a = array(6,7,8); array_push($a, 9); $a[] = 9; # same as array_push array_pop($a);	@a = (6,7,8); push @a, 9; pop @a;	a = [6,7,8] a.append(9) a.pop()	a = [6,7, a.push(9) a << 9 # a.pop

manipulate front	$a = array(6,7,8); array_unshift($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, a.unshift a.shift
concatenate	$a = array(1,2,3); $a2 = array_merge($a,array(4,5,6)); $a = array_merge($a,array(4,5,6));	@a = (1,2,3); @a2 = (@a,(4,5,6)); push @a, (4,5,6);	a = [1,2,3] a2 = a + [4,5,6] a.extend([4,5,6])	a = [1,2, a2 = a + a.concat(
replicate		@a = (undef) x 10; use Storable 'dclone'	a = [None] * 10 a = [None for i in range(0, 10)] import copy	a = [nil] a = Array
address copy, shallow copy, deep copy	$a = array(1,2,array(3,4)); $a2 =& $a; *none* $a4 = $a;	my @a = (1,2,[3,4]); my $a2 = \@a; my @a3 = @a; my @a4 = @{dclone(\@a)};	a = [1,2,[3,4]] a2 = a a3 = list(a) a4 = copy.deepcopy(a)	a = [1,2, a2 = a a3 = a.du a4 = Mars
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*
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].e
indexed iteration	$a = array("do", "re", "mi" "fa"); foreach ($a as $i => $s) { echo "$s at index $i\n"; }	*none; use range iteration from 0 to $#a and use index to look up value in the loop body*	a = ['do', 're', 'mi', 'fa'] for i, s in enumerate(a): print('%s at index %d' % (s, i))	a = %w(do a.each_wi puts "# end
iterate over range	*not space efficient; use C-style for loop*	for $i (1..1_000_000) { *code* }	range *replaces* xrange *in Python 3:* for i in xrange(1, 1000001): *code*	(1..1_000 *code* end
instantiate range as array	$a = range(1, 10);	@a = 1..10;	a = range(1, 11) *Python 3:* a = list(range(1, 11))	a = (1..1
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] a[::-1] a.reverse()	a = [1,2, a.reverse a.reverse
sort	$a = array("b", "A", "a", "B"); *none* sort($a); *none, but* usort *sorts in place*	@a = qw(b A a B); sort @a; @a = sort @a; sort { lc($a) cmp lc($b) } @a;	a = ['b', 'A', 'a', 'B'] sorted(a) a.sort() a.sort(key=str.lower)	a = %w(b a.sort a.sort! a.sort do x.downc end
dedupe	$a = array(1,2,2,3); $a2 = array_unique($a); $a = array_unique($a);	use List::MoreUtils 'uniq'; my @a = (1,2,2,3); my @a2 = uniq @a; @a = uniq @a;	a = [1,2,2,3] a2 = list(set(a)) a = list(set(a))	a = [1,2, a2 = a.un a.uniq!
membership	in_array(7, $a)	7 ~~ @a	7 in a	a.include
intersection	$a = array(1,2); $b = array(2,3,4) array_intersect($a, $b)		{1,2} & {2,3,4}	[1,2] & [
union	$a1 = array(1,2); $a2 = array(2,3,4); array_unique(array_merge($a1, $a2))		{1,2} | {2,3,4}	[1,2] | [
relative complement, symmetric difference	$a1 = array(1,2,3); $a2 = array(2); array_values(array_diff($a1, $a2)) *none*		{1,2,3} - {2} {1,2} ^ {2,3,4}	require ' [1,2,3] - Set[1,2]
map	array_map(function ($x) { return $x*$x; }, array(1,2,3))	map { $_ * $_ } (1,2,3)	map(lambda x: x * x, [1,2,3]) # or use list comprehension: [x*x for x in [1,2,3]]	[1,2,3].m
filter	array_filter(array(1,2,3), function ($x) { return $x>1; })	grep { $_ > 1 } (1,2,3)	filter(lambda x: x > 1, [1,2,3]) # or use list comprehension: [x for x in [1,2,3] if x > 1]	[1,2,3].s
reduce	array_reduce(array(1,2,3), function($x,$y) { return $x+$y; }, 0)	use List::Util 'reduce'; reduce { $x + $y } 0, (1,2,3)	# import needed in Python 3 only from functools import reduce reduce(lambda x, y: x+y, [1,2,3], 0)	[1,2,3].i
universal and existential tests	*use array_filter*	# cpan -i List::MoreUtils use List::MoreUtils qw(all any); all { $_ % 2 == 0 } (1,2,3,4) any { $_ % 2 == 0 } (1,2,3,4)	all(i%2 == 0 for i in [1,2,3,4]) any(i%2 == 0 for i in [1,2,3,4])	[1,2,3,4] [1,2,3,4]
shuffle and sample	$a = array(1, 2, 3, 4); shuffle($a);	use List::Util 'shuffle';	from random import shuffle, sample	[1, 2, 3, *Ruby 1.9:*

	array_rand($a, 2)	@a = (1, 2, 3, 4); shuffle(@a); *none*	a = [1, 2, 3, 4] shuffle(a) sample(a, 2)	[1, 2, 3,
zip	# array of 3 pairs: $a = array_map(NULL, array(1, 2, 3), array("a", "b", "c"));	# cpan -i List::MoreUtils use List::MoreUtils 'zip'; @nums = (1, 2, 3); @lets = qw(a b c); # flat array of 6 elements: @a = zip @nums, @lets;	# array of 3 pairs: a = zip([1,2,3], ['a', 'b', 'c'])	# array o a = [1,2,

dictionaries

	php	perl	python	
literal	$d = array("t" => 1, "f" => 0);	%d = (t => 1, f => 0);	d = { 't':1, 'f':0 }	d = { "t"
size	count($d)	scalar(keys %d)	len(d)	d.size d.length
lookup	$d["t"]	$d{"t"}	d['t']	d["t"]
out-of-bounds behavior	$d = array(); *evaluates as* NULL: $d["lorem"]; *adds key/value pair:* $d["lorem"] = "ipsum";	%d = (); *evaluates as* undef: $d{"lorem"}; *adds key/value pair:* $d{"lorem"} = "ipsum";	d = {} *raises* KeyError: d['lorem'] *adds key/value pair:* d['lorem'] = 'ipsum'	d = {} *evaluates* d["lorem" *adds key/* d["lorem"
is key present	array_key_exists("y", $d);	exists $d{"y"}	'y' in d	d.has_key
delete entry	$d = array(1 => "t", 0 => "f"); unset($d[1]);	%d = (1 => "t", 0 => "f"); delete $d{1};	d = {1: True, 0: False} del d[1]	d = {1 => d.delete(
from array of pairs, from even length array		@a = (1,"a",2,"b",3,"c"); %d = @a;	a = [[1,'a'], [2,'b'], [3,'c']] d = dict(a) a = [1,'a',2,'b',3,'c'] d = dict(zip(a[::2], a[1::2]))	a = [[1," d = Hash[a = [1,"a d = Hash[
merge	$d1 = array("a"=>1, "b"=>2); $d2 = array("b"=>3, "c"=>4); $d1 = array_merge($d1, $d2);	%d1 = (a=>1, b=>2); %d2 = (b=>3, c=>4); %d1 = (%d1, %d2);	d1 = {'a':1, 'b':2} d2 = {'b':3, 'c':4} d1.update(d2)	d1 = {"a" d2 = {"b" d1.merge!
invert	$to_num = array("t"=>1, "f"=>0); $to_let = array_flip($to_num);	%to_num = (t=>1, f=>0); %to_let = reverse %to_num;	to_num = {'t':1, 'f':0} # dict comprehensions added in 2.7: to_let = {v:k for k, v in to_num.items()}	to_num = to_let =
iteration	foreach ($d as $k => $v) { *code* }	while (($k, $v) = each %d) { *code* }	for k, v in d.iteritems(): *code* *Python 3:* for k, v in d.items(): *code*	d.each do *code* end
keys and values as arrays	array_keys($d) array_values($d)	keys %d values %d	d.keys() d.values() *Python 3:* list(d.keys()) list(d.values())	d.keys d.values
sort by values	asort($d); foreach ($d as $k => $v) { print "$k: $v\n"; }	foreach $k (sort { $d{$a} <=> $d{$b} } keys %d) { print "$k: $d{$k}\n"; }	from operator import itemgetter pairs = sorted(d.iteritems(), key=itemgetter(1)) for k, v in pairs: print('{}: {}'.format(k, v))	d.sort_by puts "# end
default value, computed value	$counts = array(); $counts['foo'] += 1; *extend* ArrayObject *for computed values and defaults other than zero or empty string.*	my %counts; $counts{'foo'} += 1 *define a tied hash for computed values and defaults other than zero or empty string*	from collections import defaultdict counts = defaultdict(lambda: 0) counts['foo'] += 1 class Factorial(dict): def __missing__(self, k): if k > 1: return k * self[k-1] else: return 1 factorial = Factorial()	counts =

functions

	php	perl	python	

function declaration	function add($a, $b) { return $a + $b; }	sub add { $_[0] + $_[1] } sub add { my ($a, $b) = @_; $a + $b; }	def add(a, b): return a+b	def add(a a+b end
function invocation	add(1, 2); *function names are case insensitive:* ADD(1, 2);	add(1, 2); *parens are optional:* add 1, 2;	add(1, 2)	*parens ar* *omitted w* *parameter* add(1, 2) *parens ar* add 1, 2
missing argument behavior	*set to* NULL *with warning*	*set to* undef	*raises* TypeError	*raises* Ar
default value	function my_log($x, $base=10) { return log($x)/log($base); } my_log(42); my_log(42, M_E);	sub my_log { my $x = shift; my $base = shift // 10; log($x)/log($base); } my_log(42); my_log(42, exp(1));	import math def my_log(x, base=10): return math.log(x)/math.log(base) my_log(42) my_log(42, math.e)	def my_lo Math.lo end my_log(42 my_log(42
variable number of arguments	function foo() { $arg_cnt = func_num_args(); if ($arg_cnt >= 1) { $n = func_get_arg(0); echo "first: " . $n . "\n"; } if ($arg_cnt >= 2) { $a = func_get_args(); $n = $a[$arg_cnt-1]; echo "last: " . $n . "\n"; } }	sub foo { if (@_ >= 1) { print "first: $_[0]\n"; } if (@_ >= 2) { print "last: $_[-1]\n"; } }	def foo(*a): if len(a) >= 1: print('first: ' + str(a[0])) if len(a) >= 2: print('last: ' + str(a[-1]))	def foo(* if a.si puts end if a.si puts end end
named parameters	*none*	*none*	def fequal(x, y, eps=0.01): return abs(x - y) < eps fequal(1.0, 1.001) fequal(1.0, 1.001, eps=0.1**10)	def fequa eps = o (x - y) end fequal(1. fequal(1.
pass number or string by reference	function foo(&$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, $s);	*not possible*	*not possi*
pass array or dictionary by reference	function foo(&$x, &$y) { $x[2] = 5; $y["f"] = -1; } $a = array(1,2,3); $d = array("t"=>1,"f"=>0); foo($a, $d);	sub foo { $_[0][2] = 5; $_[1]{"f"} = -1; } my @a = (1,2,3); my %d = ("t"=> 1, "f" => 0); foo(\@a, \%d);	def foo(x, y): x[2] = 5 y['f'] = -1 a = [1,2,3] d = {'t':1, 'f':0} foo(a, d)	def foo(x x[2] = y["f"] end a = [1,2, d = {"t"= foo(a, d)
return value	return *arg or* NULL	return *arg or last expression evaluated*	return *arg or* None	return *ar*
multiple return values	function first_and_second(&$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 return end x, y = fi
lambda declaration	$sqr = function ($x) { return $x * $x; };	$sqr = sub { $_[0] * $_[0] }	*body must be an expression:* sqr = lambda x: x * x	sqr = lam

lambda invocation

```php
$sqr(2)
```

```perl
$sqr->(2)
```

```python
sqr(2)
```

```
sqr.call(
sqr[2]
```

function reference

```php
$func = "add";
```

```perl
my $func = \&add;
```

```python
func = add
```

```
func = la
```

function with private state

```php
function counter()
{
  static $i = 0;
  return ++$i;
}

echo counter();
```

```perl
use feature state;

sub counter {
  state $i = 0;
  ++$i;
}

print counter() . "\n";
```

```python
# state not private:
def counter():
  counter.i += 1
  return counter.i

counter.i = 0
print(counter())
```

none

closure

```php
function make_counter()
{
  $i = 0;
  return function () use (&$i) {
    return ++$i;
  };
}

$nays = make_counter();
echo $nays();
```

```perl
sub make_counter {
  my $i = 0;
  return sub { ++$i };
}

my $nays = make_counter;
print $nays->() . "\n";
```

```python
# Python 3:
def make_counter():
  i = 0
  def counter():
    nonlocal i
    i += 1
    return i
  return counter

nays = make_counter()
```

```
def make_
  i = 0
  return
end

nays = ma
puts nays
```

generator

none

none

```python
def make_counter():
  i = 0
  while True:
    i += 1
    yield i

nays = make_counter()
print(nays.next())
```

```
# Ruby 1.
def make_
  return
    i = 0
    while
      i +
      Fib
    end
  end
end

nays = ma
puts nays
```

decorator

```python
def logcall(f):
  def wrapper(*a, **opts):
    print('calling ' + f.__name__)
    f(*a, **opts)
    print('called ' + f.__name__)
  return wrapper

@logcall
def square(x):
  return x * x
```

operator as function

```python
import operator

operator.mul(3, 7)

a = ['foo', 'bar', 'baz']
operator.itemgetter(2)(a)
```

```
3.*(7)

a = ['foo
a.[](2)
```

execution control

	php	perl	python	
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 == $n ) {` `  print "one hit\n"` `} else {` `  print "$n hits\n"` `}`	`if 0 == n:` `  print('no hits')` `elif 1 == n:` `  print('one hit')` `else:` `  print(str(n) + ' hits')`	`if n == 0` `  puts "n` `elsif 1 =` `  puts "o` `else` `  puts "#` `end`
switch	`switch ($n) {` `case 0:` `  echo "no hits\n";` `  break;` `case 1:` `  echo "one hit\n";` `  break;` `default:` `  echo "$n hits\n";` `}`	`use feature 'switch';` `given ($n) {` `  when (0) { print "no hits\n"; }` `  when (1) { print "one hit\n"; }` `  default { print "$n hits\n"; }` `}`	*none*	`case n` `when 0` `  puts "n` `when 1` `  puts "o` `else` `  puts "#` `end`
while	`while ( $i < 100 ) { $i++; }`	`while ( $i < 100 ) { $i++ }`	`while i < 100:` `  i += 1`	`while i <` `  i += 1` `end`
c-style for	`for ($i = 1; $i <= 10; $i++) {` `  echo "$i\n";` `}`	`for ( $i=0; $i <= 10; $i++ ) {` `  print "$i\n";` `}`	*none*	*none*

break, continue, redo	break continue *none*	last next redo	break continue *none*	break nex
control structure keywords	case default do else elseif for foreach goto if switch while	do else elsif for foreach goto if unless until while	elif else for if while	case do e unless un
what do does	*starts body of a* do-while *loop, a loop which checks the condition after the body is executed*	*executes following block and returns value of last statement executed*	*raises* NameError *unless a value was assigned to it*	*starts an body of a*
statement modifiers	*none*	print "positive\n" if $i > 0; print "nonzero\n" unless $i == 0;	*none*	puts "pos puts "non
raise exception	throw new Exception("bad arg");	die "bad arg";	raise Exception('bad arg')	# raises raise "ba
re-raise exception			try: raise Exception('bam!') except: print('re-raising...') raise	begin raise " rescue puts "r raise end
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 begin risky rescue print " puts $! end
global variable for last exception	*none*	*$EVAL_ERROR: $@* *$OS_ERROR: $!* *$CHILD_ERROR: $?*	*last exception:* sys.exc_info()[1]	*last exce* *backtrace* *exit stat*
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 def ini super end end
catch exception by type	try { throw new Bam; } catch (Bam $e) { echo $e->getMessage(), "\n"; }	*none*	try: raise Bam() except Bam as e: print(e)	begin raise B rescue Ba puts e. end
finally/ensure	*none*	*none*	acquire_resource() try: risky() finally: release_resource()	acquire_r begin risky ensure release end
start thread	*none*	use threads; $func = sub { sleep 10 }; $thr = threads->new($func);	class sleep10(threading.Thread): def run(self): time.sleep(10) thr = sleep10() thr.start()	thr = Thr
wait on thread		$thr->join;	thr.join()	thr.join

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

General [link] ¶

versions used [link] ¶

The versions used for testing code in the reference sheet.

implicit prologue [link] ¶

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

perl:

We adopt the convention that if an example uses a variable without declaring it, it should be taken to have been previously declared with my.

python:

To keep the examples short we assume that os, re, and sys are always imported.

show version [link] ¶

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) or RUBY_VERSION (Ruby 1.9).

Grammar and Invocation [link] ¶

interpreter [link] ¶

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 [link] ¶

The customary name of the repl.

php:

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

php -a offers a different style of interactive mode. It collects input until EOF is encountered and then it executes it. Text inside <? *code* ?> and <?= *code* ?> is executed as PHP code. Text outside of PHP markup tags is echoed.

perl:

The Perl REPL perl -de 0 does not save or display the result of an expression. perl -d is the Perl debugger and perl -e runs code provided on the command line.

perl -de 0 does not by default have readline, but it can be added:

```
$ cpan -i Term::Readline::Perl

  <cpan output omitted>

$ perl -de 0

  DB<1> use Term::Readline::Perl;

  DB<2> print 1 + 1;
2
```

python:

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

ruby:

irb saves the result of the last statement in _.

command line program [link] ¶

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

block delimiters [link] ¶

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 encounters a tab, it is treated as 8 spaces.

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

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

ruby:

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

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

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

statement separator [link] ¶

How the parser determines the end of a statement.

php:

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

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

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

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

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.

end-of-line comment [link] ¶

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

multiple line comment [link] ¶

How to comment out multiple lines.

python:

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

Variables and Expressions [link] ¶

local variable [link] ¶

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.

python:

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

ruby:

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

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

regions which define lexical scope [link] ¶

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

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

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

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

php:

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

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

Local variables cannot be created in class bodies.

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 local 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. Python trivia question: what would happen if the following code were executed?

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

foo()
```

ruby:

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 [link] ¶

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

ruby:

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

constant [link] ¶

How to declare a constant.

php:

A constant can be declared inside a class:

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

Refer to a class constant like this:

```
Math::pi
```

ruby:

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

assignment [link] ¶

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

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

```
a = b = 3
```

ruby:

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

parallel assignment [link] ¶

How to assign values to variables in parallel.

python:

The r-value can be a list or tuple:

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

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

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

This assignment will raise a TypeError:

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

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

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

ruby:

The r-value can be an array:

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

swap [link] ¶

How to swap the values held by two variables.

compound assignment [link] ¶

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

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

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

The compound assignment operators are displayed in this order:

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

python:

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

increment and decrement [link] ¶

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

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

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

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

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

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

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

php:

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

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

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.

null [link] ¶

The null literal.

null test [link] ¶

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:

```
$v = undef;
if ($v == undef) { print "true"; }

$v = 0;
if ($v == undef) { print "sadly true"; }

$v = '';
if ($v == undef) { print "sadly true"; }
```

undefined variable access [link] ¶

The result of attempting to access an undefined variable.

php:

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

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

perl:

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.

python:

How to test if a variable is defined:

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

ruby:

How to test if a variable is defined:

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

conditional expression [link] ¶

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

condition ? *true value* : *false value*

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

python:

The Python conditional expression comes from Algol.

ruby:

The Ruby if statement is also an expression:

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

Arithmetic and Logic [link] ¶

true and false [link] ¶

Literals for the booleans.

These are the return values of the relational operators.

php:

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

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

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

falsehoods [link] ¶

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

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

python:

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

logical operators [link] ¶

Logical and, or, and not.

php, perl, ruby:

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

relational operators [link] ¶

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

php:

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

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*.

python:

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

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

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

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

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

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

min and max [link] ¶

How to get the min and max.

three value comparison [link] ¶

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

The <=> symbol is called the spaceship operator.

arithmetic operators [link] ¶

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

integer division [link] ¶

How to get the integer quotient of two integers. How to get the integer quotient and remainder.

perl:

The integer pragma makes all arithmetic operations integer operations. Floating point numbers are truncated before they are used. Hence integer division could be performed with:

```
use integer;
my $a = 7 / 3;
no integer;
```

divmod [link] ¶

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

integer division by zero [link] ¶

What happens when an integer is divided by zero.

float division [link] ¶

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

float division by zero [link] ¶

What happens when a float is divided by zero.

power [link] ¶

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

sqrt [link] ¶

The square root function.

sqrt -1 [link] ¶

The result of taking the square root of negative one.

transcendental functions [link] ¶

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

python:

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

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

ruby:

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

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

transcendental constants [link] ¶

Constants for π and Euler's constant.

float truncation [link] ¶

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

perl:

The CPAN module Number::Format provides a round function. The 2nd argument specifies the number of digits to keep to the right of the radix. The default is 2.

```
use Number::Format 'round';

round(3.14, 0);
```

absolute value [link] ¶

How to get the absolute value of a number.

integer overflow [link] ¶

What happens when the largest representable integer is exceeded.

float overflow [link] ¶

What happens when the largest representable float is exceeded.

rational numbers [link] ¶

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

ruby:

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

complex numbers [link] ¶

python:

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

random integer, uniform float, normal float [link] ¶

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

set random seed, get and restore seed [link] ¶

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

All the languages in the sheet set the seed automatically to a value that is difficult to predict. The Ruby 1.9 MRI interpreter uses the current time and process ID, for example. As a result there is usually no need to set the seed.

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

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

bit operators [link] ¶

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

binary, octal, and hex literals [link] ¶

Binary, octal, and hex integer literals

base conversion [link] ¶

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

perl

Perl has the functions oct and hex which convert strings encoded in octal and hex and return the corresponding integer. The oct function will handle binary or hex encoded strings if they have "0b" or "0x" prefixes.

```
oct("60")
oct("060")
oct("0b101010")
oct("0x2a")

hex("2a")
hex("0x2a")
```

python

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

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

Strings [link] ¶

string literal [link] ¶

The syntax for string literals.

perl:

When use strict is not in effect bareword strings are permitted.

Barewords are strings without quote delimiters. They are a feature of shells. Barewords cannot contain whitespace or any other character used by the tokenizer to distinguish words.

Before Perl 5 subroutines were invoked with an ampersand prefix & or the older do keyword. With Perl 5 neither is required, but this made it impossible to distinguish a bareword string from a subroutine without knowing all the subroutines which are in scope.

The following code illustrates the bareword ambiguity:

```
no strict;

print rich . "\n";  # prints "rich"; rich is a bareword string

sub rich { return "poor" }

print rich . "\n";  # prints "poor"; rich is now a subroutine
```

newline in literal [link] ¶

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"""
```

character escapes [link] ¶

Backslash escape sequences for inserting special characters into string literals.

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

perl:

In addition to the character escapes, Perl has the following translation escapes:

\u make next character uppercase
\l make next character lowercase
\U make following characters uppercase
\L make following characters lowercase
\Q backslash escape following nonalphanumeric characters
\E end \U, \L, or \Q section

When use charnames is in effect the \N escape sequence is available:

```
binmode(STDOUT, ':utf8');

use charnames ':full';

print "lambda: \N{GREEK SMALL LETTER LAMDA}\n";

use charnames ':short';

print "lambda: \N{greek:lamda}\n";

use charnames qw(greek);

print "lambda: \N{lamda}\n";
```

python:

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

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

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

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

variable interpolation [link] ¶

How to interpolate variables into strings.

python:

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

A selection of variables in scope can be passed explicitly:

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

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

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

custom delimiters [link] ¶

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

sprintf [link] ¶

How to create a string using a printf style format.

python:

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

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

The curly braces are escaped by doubling:

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

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

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

here document [link] ¶

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 variables into strings.
EOF
```

concatenate [link] ¶

The string concatenation operator.

replicate [link] ¶

The string replication operator.

split [link] ¶

How to split a string containing a separator into an array of substrings; how to split a string in two; how to split a string with the delimiters preserved as separate elements; how to split a string into an array of single character strings.

See also scan.

python:

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

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

join [link] ¶

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

case manipulation [link] ¶

How to put a string into all caps or all lower case letters. How to capitalize the first letter of a string.

trim [link] ¶

How to remove whitespace from the ends of a string.

perl:

An example of how to trim a string without installing a library:

```
$s = " lorem ";
$s =~ s/^\s*(.*)\s*$/\1/;
```

The return value of the =~ operator is boolean, indicating whether a match occurred. Also the left hand side of the =~ operator must be a scalar variable that can be modified. Using the =~ operator is necessarily imperative, unlike the Text::Trim functions which can be used in expressions.

pad on right, on left [link] ¶

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

convert from string, to string [link] ¶

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.

length [link] ¶

How to get the length in characters of a string.

index of substring [link] ¶

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

python:

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

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

extract substring [link] ¶

How to extract a substring from a string by index.

extract character [link] ¶

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

chr and ord [link] ¶

Converting characters to ASCII codes and back.

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

character translation [link] ¶

How to apply a character mapping to a string.

Regular Expressions [link] ¶

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

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

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

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

character classes: [ ] .

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

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

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

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

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

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

character class abbreviations:

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

alternation and grouping: | ( )

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

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

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

quantifiers: * + ? { }

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

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

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

anchors: ^ $

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

**escaping: **

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

php:

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

literal, custom delimited literal [link] ¶

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

php:

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

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

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

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

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

int preg_match_all ( string $pattern , string $subject [, array &$matches [, int $flags = PREG_PATTERN_ORDER [, int $offset = 0 ]]] )
```

python:

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

Compiling regexes can always be avoided:

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

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

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

character class abbreviations and anchors [link] ¶

The supported character class abbreviations and anchors.

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

match test [link] ¶

How to test whether a string matches a regular expression.

python:

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

ruby:

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

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

and

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

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

case insensitive match test [link] ¶

How to perform a case insensitive match test.

modifiers [link] ¶

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

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

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

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

substitution [link] ¶

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

php:

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

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

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

perl:

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

The g modifiers specifies that all occurrences should be replaced. If omitted, only the first occurrence is replaced.

python:

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

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

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

ruby:

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

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

match, prematch, postmatch [link] ¶

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

perl:

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

ruby:

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

group capture [link] ¶

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

ruby:

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

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

scan [link] ¶

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

backreference in match and substitution [link] ¶

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

recursive regex [link] ¶

Examples of recursive regexes.

The examples match substrings containing balanced parens.

Date and Time [link] ¶

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

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

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

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

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

Perl and Python both use and expose the tm struct of the standard library. In the case of Perl, the first nine values of the struct (up to the member tm_isdst) are put into an array. Python, meanwhile, has a module called time which is a thin wrapper to the standard library functions which operate on this struct. Here is how get a tm struct in Python:

```
import time

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

The tm struct is a low level entity, and interacting with it directly should be avoided. In the case of Python it is usually sufficient to use the datetime module instead. For Perl, one can use the Time::Piece module to wrap the tm struct in an object.

date/time type [link] ¶

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

perl

Built in Perl functions work with either (1) scalars containing the Unix epoch as an integer or (2) arrays containing the first nine values of the standard C library tm struct. When use Time::Piece is in effect functions which work with tm arrays are replaced with variant that work with the Time::Piece wrapper.

The modules Time::Local and Date::Parse can create scalars containing the Unix epoch.

CPAN provides the DateTime module which provides objects with functionality comparable to the DateTime objects of PHP and Python.

current date/time [link] ¶

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

to unix epoch, from unix epoch [link] ¶

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

python:

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

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

```
import calendar
import datetime

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

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

current unix epoch [link] ¶

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

strftime [link] ¶

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

php:

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

default format example [link] ¶

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

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

php:

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

strptime [link] ¶

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

parse date w/o format [link] ¶

How to parse a date without providing a format string.

result date subtraction [link] ¶

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

add time duration [link] ¶

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

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

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

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

local timezone [link] ¶

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

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

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

php:

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

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

Here is the list of timezones supported by PHP.

arbitrary timezone [link] ¶

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

timezone name, offset from UTC, is daylight savings? [link] ¶

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

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

perl:

It is not necessary to create a DateTime object to get the local timezone offset:

```
use Time::Piece;

$t = localtime(time);
$offset_hrs = $t->tzoffset / 3600;
```

ruby:

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

microseconds [link] ¶

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

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

sleep [link] ¶

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

php:

PHP provides usleep which takes an argument in microseconds:

```
usleep(500000);
```

perl:

The Perl standard library includes a version of sleep which supports fractional seconds:

```
use Time::HiRes qw(sleep);

sleep 0.5;
```

timeout [link] ¶

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

Techniques relying on SIGALRM only work on Unix systems.

Arrays [link] ¶

What the languages call their basic container types:

	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 dictionaries.

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.

literal [link] ¶

Array literal syntax.

perl:

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

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

quote words [link] ¶

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

size [link] ¶

How to get the number of elements in an array.

empty test [link] ¶

How to test whether an array is empty.

lookup [link] ¶

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.

update [link] ¶

How to update the value at an index.

out-of-bounds behavior [link] ¶

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

index of array element [link] ¶

perl:

Some techniques for getting the index of an array element.

slice by endpoints, by length [link] ¶

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

perl:

Perl arrays can take an array of indices as the index value. The range of values selected can be discontinuous and the order of the values can be manipulated:

```
@nums = (1,2,3,4,5,6);
@nums[(1,3,2,4)];
```

python:

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

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

Python has notation for taking every nth element:

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

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

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

slice to end [link] ¶

How to slice to the end of an array.

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

manipulate back [link] ¶

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

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

manipulate front [link] ¶

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

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

concatenate [link] ¶

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

replicate [link] ¶

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

address copy, shallow copy, deep copy [link] ¶

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 in that different syntax has to be used to access the original array and the address copy: @a and @$a1. To make @a1 and @a refer to the same array, use typeglobs:

```
*a1 = *a;
```

python:

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

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

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

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

arrays as function arguments [link] ¶

How arrays are passed as arguments.

iteration [link] ¶

How to iterate through the elements of an array.

perl:

for and foreach are synonyms. Some use for exclusively for C-style for loops and foreach for array iteration.

indexed iteration [link] ¶

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

iterate over range [link] ¶

Iterate over a range without instantiating it as a list.

perl:

With Perl 5.005 the for and foreach operators were optimized to not instantiate a range argument as a list.

instantiate range as array [link] ¶

How to convert a range to an array.

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

python:

In Python 2 range() returns a list.

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

ruby:

The Range class includes the Enumerable module.

reverse [link] ¶

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

python:

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

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

reversed can be used to create a reversed list:

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

sort [link] ¶

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

php:

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

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

$a = array("b", "A", "a", "B");

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

dedupe [link] ¶

How to remove extra occurrences of elements from an array.

python:

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

membership [link] ¶

How to test for membership in an array.

intersection [link] ¶

How to compute an intersection.

python:

Python has literal notation for sets:

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

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

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

ruby:

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

union [link] ¶

ruby:

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

relative complement, symmetric difference [link] ¶

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

ruby:

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

map [link] ¶

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

ruby:

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

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

filter [link] ¶

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

ruby:

The in place version is select!.

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

reduce [link] ¶

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

python:

reduce is not needed to sum a list of numbers:

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

ruby:

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

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

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

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

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

universal and existential tests [link] ¶

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

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

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

shuffle and sample [link] ¶

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

php:

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

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

zip [link] ¶

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

perl:

zip expects arrays as arguments, which makes it difficult to define the arrays to be zipped on the same line as the invocation. It can be done like this:

```
@a = zip @{[1,2,3]}, @{['a','b','c']};
```

Dictionaries [link] ¶

literal [link] ¶

perl:

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

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

size [link] ¶

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

lookup [link] ¶

How to lookup a dictionary value using a dictionary key.

perl:

Use the ampersand prefix @ to slice a Perl hash. The index is a list of keys.

```
%nums = ('b'=>1, 't'=>2, 'a'=>3);
@nums{('b','t')}
```

out-of-bounds behavior [link] ¶

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

python:

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

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

is key present [link] ¶

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

delete entry [link] ¶

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

from array of pairs, from even length array [link] ¶

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

merge [link] ¶

How to merge the values of two dictionaries.

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

invert [link] ¶

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

iteration [link] ¶

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

python:

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

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

keys and values as arrays [link] ¶

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

python:

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

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

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

sort by values [link] ¶

How to sort the data in a dictionary by its values.

default value, computed value [link] ¶

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

php:

Extend ArrayObject to compute values on lookup:

```
class Factorial extends ArrayObject {

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

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

$factorial = new Factorial();
```

perl:

How to use a tied hash. If the CPAN module Tie::ExtraHash is installed there is a shorter way.

Functions [link] ¶

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 [link] ¶

How to define a function.

perl:

One can also use shift to put the arguments into local variables:

```
sub add {
  my $a = shift;
  my $b = shift;

  $a + $b;
}
```

function invocation [link] ¶

How to invoke a function.

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 [link] ¶

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 [link] ¶

How to declare a default value for an argument.

variable number of arguments [link] ¶

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

python:

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

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

ruby:

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

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

named parameters [link] ¶

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

python:

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

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

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

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

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

In Python 3 named parameters can be made mandatory:

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

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

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

pass number or string by reference [link] ¶

How to pass numbers or strings by reference.

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

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

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

perl:

Here is a potential for confusion: if a reference is used in Perl to pass data, that is *pass by address*, not *pass by reference*. A Perl reference is comparable to a pointer in C, albeit one that knows the data type of what it points to at runtime. C++ has both pointers and references and thus can pass data by address or by reference, though pass by value is the default.

pass array or dictionary by reference [link] ¶

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

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 [link] ¶

How the return value of a function is determined.

multiple return values [link] ¶

How to return multiple values from a function.

lambda declaration and invocation [link] ¶

How to define and invoke a lambda function.

python:

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

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

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

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

Python closures are read only.

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

ruby:

The following lambda and Proc object behave identically:

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

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

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

Ruby 1.9 introduces new syntax for defining lambdas and invoking them:

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

sqr.(2)
```

function reference [link] ¶

How to store a function in a variable.

php:

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

python:

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

function with private state [link] ¶

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

python:

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

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

print(counter())
```

closure [link] ¶

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

python:

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

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

generator [link] ¶

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

perl:

CPAN provides a module called Coro which implements coroutines. Some notes on the distinction between coroutines and generators.

python:

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

ruby:

Ruby generators are called fibers.

decorator [link] ¶

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

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

operator as function [link] ¶

How to call an operator using the function invocation syntax.

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

python:

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

ruby:

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

Execution Control [link] ¶

if [link] ¶

The if statement.

php:

PHP has the following alternate syntax for if statements:

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

perl:

When an if block is the last statement executed in a subroutine, the return value is the value of the branch that executed.

ruby:

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

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

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

switch [link] ¶

The switch statement.

while [link] ¶

php:

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

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

perl:

Perl provides until, do-while, and do-until loops.

An until or a do-until loop can be replaced by a while or a do-while loop by negating the condition.

ruby:

Ruby provides a loop with no exit condition:

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

Ruby also provides the until loop.

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

c-style for [link] ¶

How to write a C-style for loop.

break, continue, redo [link] ¶

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

control structure keywords [link] ¶

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

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

what do does [link] ¶

How the do keyword is used.

perl:

The do keyword can convert an if statement to a conditional expression:

```
my $m = do {
  if ($n) { 1 }
  else { 0 }
};
```

statement modifiers [link] ¶

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

Perl and Ruby have conditional statement modifers. Ruby also has looping statement modifers.

ruby:

Ruby has the looping statement modifiers while and until:

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

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

raise exception [link] ¶

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.

re-raise exception [link] ¶

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

python:

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

ruby:

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

catch exception [link] ¶

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 [link] ¶

The global variable name for the last exception raised.

define exception [link] ¶

How to define a new variable class.

catch exception by type [link] ¶

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 [link] ¶

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

start thread [link] ¶

ruby:

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

wait on thread [link] ¶

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

PHP [link] ¶

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

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

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

- PHP CGI/SAPI writes HTTP headers to standard out before any output specified by the program. PHP CLI does not.

- PHP CLI sets the constants STDIN, STDOUT, and STDERR. PHP CGI/SAPI do not.
- PHP CLI has no timeout. PHP CGI/SAPI will typically timeout a script after 30 seconds.
- PHP CGI/SAPI add HTML markup to error messages. PHP CLI does not.
- PHP CLI does not buffer output, so calling flush is never necessary. PHP CGI/SAPI buffer output.

Perl [link] ¶

perldoc
core modules
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 a scalar to an array will modify 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 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 the => operator 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 keyword 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 regex matched, part of the string before and after the match
- $1 ⋯ $9: what subpatterns in last regex matched

Python [link] ¶

2.7: Language, Standard Library
Why Python3 Summary of Backwardly Non-compatible Changes in Python 3
3.2: Language, Standard Library
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}. This notation is also available in Python 2.7. Dictionaries and sets are implemented using hash tables and as a result dictionary keys and set elements must be hashable.

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

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

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

Ruby [link] ¶

1.8.7 core, stdlib
1.9.3 core, stdlib
ruby-style-guide bbatsov

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

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

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

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

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

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

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

Ruby blocks are closures like lambda functions and can see local variables in the enclosing scope in which they were defined. In Ruby 1.8 the arguments of the block are local to the block unless the variables were already in use in the enclosing scope. In Ruby 1.9 this was changed so that block arguments are always local to the block. In addition Ruby 1.9 add semicolon syntax so that identifiers listed after the arguments could be made local to the block even if already defined in the containing scope.

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

Testimonials

""We went live on budget, to specification and on time"" -- ajt

""Perl saved my vacation!"" -- Tom Moertel

More⋯

News

- 31-May-2015: New Pages and updated content
- 09-Jul-2012: Perl Humour page, #perl FAQ, and other new pages
- 22-Jul-2011: The book Modern Perl, exercises and challenges, and some new topical pages
- More news...

Webmaster: Shlomi Fish (Email - shlomif@shlomifish.org)

Original Design: GoFlexiblePro | Author: G. Wolfgang | W3C XHTML5 | W3C CSS 3

Hosted by: Hexten.net.