

Raku (programming language)

Raku is a member of the Perl family of programming languages.^[6] Formerly known as **Perl 6**, it was renamed in October 2019.^{[7][8]} Raku introduces elements of many modern and historical languages. Compatibility with Perl was not a goal, though a compatibility mode is part of the specification. The design process for Raku began in 2000.

Contents

History

Initial goals and implications

Mascot

Implementations

Historical implementations

Module system

Major changes from Perl

A specification

A type system

Formal subroutine parameter lists

Parameter passing modes

Blocks and closures

Sigil invariance

Object-oriented programming

Inheritance, Roles and Classes

Regular expressions

Syntactic simplification

Chained comparisons

Lazy evaluation

Gather

Junctions

Macros

Identifiers

Examples

Hello world

Factorial

Quicksort

Tower of Hanoi

Books

Books published before Perl 6 version 1.0 (known as version 6.c)

Raku



Camelia, the Raku mascot^[1]

Paradigm	<u>Multi-paradigm</u>
Family	<u>Perl</u>
Designed by	<u>Larry Wall</u>
Developer	<u>Raku community</u>
First appeared	<u>25 December 2015</u>
Stable release	<u>6.d 'Diwali' ^[2] / 24 October 2020</u>
Typing discipline	<u>Dynamic, gradual</u>
OS	<u>Cross-platform</u>
License	<u>GNU General Public License or Artistic License 2</u>
Filename extensions	<u>.p6, .pm6, .pod6, .t6, .raku, .rakumod, .rakudoc, .rakutest ^[3]</u>
Website	<u>raku.org (http://raku.org)</u>
Major implementations	
<u>Rakudo</u>	
Influenced by	
<u>Perl, Ruby, Smalltalk, Haskell^[4], JavaScript</u>	
Influenced	
<u>Perl,^[5] Haskell,^[5] AntLang</u>	

Books published after Perl 6 version 1.0 (known as version 6.c)

Books published with the new Raku name

Books to be published

References

External links

History

In Perl 6, we decided it would be better to fix the language than fix the user.

— Larry Wall^[9]

The Raku design process was first announced on 19 July 2000, on the fourth day of that year's Perl Conference,^[10] by Larry Wall in his *State of the Onion 2000* talk.^[11] At that time, the primary goals were to remove "historical warts" from the language; "easy things should stay easy, hard things should get easier, and impossible things should get hard"; a general cleanup of the internal design and APIs. The process began with a series of requests for comments or "RFCs". This process was open to all contributors, and left no aspect of the language closed to change.^[12]

Once the RFC process was complete, Wall reviewed and classified each request (361 were received). He then began the process of writing several "Apocalypses", a term which means "revealing".^[13] While the original goal was to write one Apocalypse for each chapter of *Programming Perl*, it became obvious that, as each Apocalypse was written, previous Apocalypses were being invalidated by later changes. For this reason, a set of Synopses were published, each one relating the contents of an Apocalypse, but with any subsequent changes reflected in updates. Today, the Raku specification is managed through the "roast" testing suite^[14], while the Synopses are kept as a historical reference.^[15]

There are also a series of Exegeses written by Damian Conway that explain the content of each Apocalypse in terms of practical usage. Each Exegesis consists of code examples along with discussion of the usage and implications of the examples.^[16]

There are three primary methods of communication used in the development of Raku today. The first is the #raku connect (<https://webchat.freenode.net/?channels=#raku>) IRC channel on freenode. The second is a set of mailing lists on The Perl Foundation's servers at perl.org (<http://perl.org>).^[17] The third is the Git source code repository hosted at <https://github.com/raku>.

Initial goals and implications

The major goal Wall suggested in his initial speech was the removal of historical warts. These included the confusion surrounding sigil usage for containers; the ambiguity between the select functions; the syntactic impact of bareword filehandles. There were many other problems that Perl programmers had discussed fixing for years and these were explicitly addressed by Wall in his speech.

An implication of these goals was that Perl 6 would not have backward compatibility with the existing Perl codebase. This meant that some code which was correctly interpreted by a Perl 5 compiler would not be accepted by a Perl 6 compiler. Since backward compatibility is a common goal when enhancing software, the

breaking changes in Perl 6 had to be stated explicitly. The distinction between Perl 5 and Perl 6 became so large that eventually Perl 6 was renamed Raku.

Over the years, Raku has undergone several alterations in its direction. The introduction of concepts from Python and Ruby were early influences. Furthermore, since Pugs - the first interpreter able to run Raku - was written in the Haskell programming language, many functional programming influences were absorbed by the Raku design team.

Mascot

The language's mascot is "Camelia, the Raku bug".^[1] Her name is a nod to the camel mascot associated with Perl, and her form, in the pun-loving tradition of the Perl community, is a play on "software bug". Spiral designs embedded in her butterfly-like wings resemble the characters "P6", the favored nickname for Perl 6, and off-center eye placement is an intentional pun on "Wall-eyed".^[18]

One of the goals behind the lively and colorful design of the logo was to discourage misogyny in the community and for it to be an opportunity for those of "masculine persuasion" to show their sensitive side.^[19]



Larry Wall and Camelia

Implementations

As of 2017, only the Rakudo implementation is under active development. No implementation will be designated as the official Raku implementation; rather, "Raku is anything that passes the official test suite."^[20]

Rakudo Perl 6^{[21][22]} targets a number of virtual machines, such as MoarVM, the Java Virtual Machine, and JavaScript. MoarVM is a virtual machine built especially for Rakudo^[23] and the NQP Compiler Toolchain.^[24] There is a layer between Raku and the virtual machines called Not Quite Perl 6, or NQP, which implements Raku rules for parsing Raku, as well as an abstract syntax tree and backend-specific code generation. Large portions of Rakudo are written in Raku itself, or in its subset NQP. Rakudo is not a completely self-hosting implementation, nor are there concrete plans at this point to make Rakudo a bootstrapping compiler.

Historical implementations

Pugs was an initial implementation of Perl 6 written in Haskell. Pugs used to be the most advanced implementation of Perl 6, but since mid 2007 it is mostly dormant (with updates made only to track the current version of GHC). As of November 2014 Pugs was not being actively maintained.^[25]

In 2007, v6-MiniPerl6 ("mp6") and its reimplement, v6-KindaPerl6 ("kp6") were written as a means to bootstrap the Perl-6.0.0 STD, using Perl 5. The STD is a full grammar for Perl 6 and is written in Perl 6. In theory, anything capable of parsing the STD and generating executable code is a suitable bootstrapping system for Perl 6. kp6 is currently compiled by mp6 and can work with multiple backends.^{[26][27]} mp6 and kp6 are not full Perl 6 implementations and are designed only to implement the minimum featureset required to bootstrap a full Perl 6 compiler.

Yapsi is a Perl 6 compiler and runtime written in Perl 6 itself. As a result, it requires an existing Perl 6 interpreter, such as one of the Rakudo Star releases, in order to run.^[28]

Niecza, another major Perl 6 implementation effort, focused on optimization and efficient implementation research. It targets the Common Language Infrastructure.^[29]

Module system

The Raku specification requests that modules be identified by name, version, and authority.^[30] It is possible to load only a specific version of a module, or even two modules of the same name that differ in version or authority. As a convenience, aliasing to a short name is provided.

CPAN, the Perl module distribution system, does not yet handle Raku modules. Instead a prototype module system is in use.^[31]

Major changes from Perl

Perl and Raku differ fundamentally, though in general the intent has been to "keep Raku Perl", so that Raku is clearly "a Perl programming language". Most of the changes are intended to normalize the language, to make it easier for novice and expert programmers alike to understand, and to make "easy things easier and hard things more possible".

A specification

A major non-technical difference between Perl and Raku is that Raku began as a specification.^[20] This means that Raku can be re-implemented if needed, and it also means that programmers do not have to read the source code for the ultimate authority on any given feature. The official documentation is not considered authoritative and only describes the behavior of the actual Perl interpreter informally. Any discrepancies found between the documentation and the implementation may lead to either being changed to reflect the other, a dynamic which drives the continuing development and refinement of the Perl releases.

A type system

In Raku, the dynamic type system of Perl has been augmented by the addition of static types.^[32] For example:

```
my Int $i = 0;
my Rat $r = 3.142;
my Str $s = "Hello, world";
```

However, static typing remains optional, so programmers can do most things without any explicit typing at all:

```
my $i = "25" + 10; # $i is 35
```

Raku offers a gradual typing system, whereby the programmer may choose to use static typing, use dynamic typing, or mix the two.

Formal subroutine parameter lists

Perl defines subroutines without formal parameter lists at all (though simple parameter counting and some very loose type checking can be done using Perl's "prototypes"). Subroutine arguments passed in are aliased into the elements of the array `@_`. If the elements of `@_` are modified, the changes are reflected in the original data.

Raku introduces true formal parameters to the language.^[33] In Raku, a subroutine declaration looks like this:

```
sub do_something(Str $thing, Int $other) {  
    ...  
}
```

As in Perl, the formal parameters (i.e., the variables in the parameter list) are aliases to the actual parameters (the values passed in), but by default, the aliases are constant so they cannot be modified. They may be declared explicitly as read-write aliases for the original value or as copies using the `is rw` or `is copy` directives respectively should the programmer require them to be modified locally.

Parameter passing modes

Raku provides three basic modes of parameter passing: positional parameters, named parameters, and slurpy parameters.

Positional parameters are the typical ordered list of parameters that most programming languages use. All parameters may also be passed by using their name in an unordered way. Named-only parameters (indicated by a `:` before the parameter name) can only be passed by specifying its name, i.e. it never captures a positional argument. Slurpy parameters (indicated by an `*` before the parameter name) are Raku's tool for creating variadic functions. A slurpy hash will capture remaining passed-by-name parameters, whereas a slurpy array will capture remaining passed-by-position parameters.

Here is an example of the use of all three parameter-passing modes:

```
sub somefunction($a, $b, :$c, :$d, *@e) {  
    ...  
}  
  
somefunction(1, 2, :d(3), 4, 5, 6); # $a=1, $b=2, $d=3, @e=(4,5,6)
```

Positional parameters, such as those used above, are always required unless followed by `?` to indicate that they are optional. Named parameters are optional by default, but may be marked as required by adding `!` after the variable name. Slurpy parameters are *always* optional.

Blocks and closures

Parameters can also be passed to arbitrary blocks, which act as closures. This is how, for example, `for` and `while` loop iterators are named. In the following example, a list is traversed, 3 elements at a time, and passed to the loop's block as the variables, `$a`, `$b`, `$c`.^[34]

```
for @list -> $a, $b, $c {  
    ...  
}
```

This is generally referred to as a "pointy sub" or "pointy block", and the arrow behaves almost exactly like the `sub` keyword, introducing an anonymous closure (or anonymous subroutine in Perl terminology).^[33]

Sigil invariance

In Perl, sigils – the punctuation characters that precede a variable name – change depending on how the variable is used:

```
# Perl code
my @array = ('a', 'b', 'c');
my $element = $array[1];      # $element equals 'b',
my @extract = @array[1, 2];   # @extract equals ('b', 'c')
my $element = @array[1];      # 'b' comes with a warning (5.10 option)
```

In Raku, sigils are invariant, which means that they do not change based on whether it is the array or the array element that is needed:^[32]

```
# Raku code
my @array = 'a', 'b', 'c';
my $element = @array[1];      # $element equals 'b'
my @extract = @array[1];      # @extract equals ('b')
my @extract = @array[1, 2];   # @extract equals ('b', 'c')
```

The variance in Perl is inspired by number agreement in English and many other natural languages:

"This apple."	# \$a	CORRECT
"These apples."	# @a	CORRECT
"This third apple."	# \$a[3]	CORRECT
"These third apple."	# @a[3]	WRONG

However, this conceptual mapping breaks down when references come into play, since they may refer to data structures even though they are scalars. Thus, dealing with nested data structures may require an expression of both singular and plural form in a single term:

```
# Perl code: retrieve a list from the leaf of a hash containing hashes that contain arrays
my @trans_verbs = @{ $dictionary{ 'verb' }{ 'transitive' } };
```

This complexity has no equivalent either in common use of natural language or in other programming languages, and it causes high cognitive load when writing code to manipulate complex data structures. This is the same code in Raku:

```
# Raku code: retrieve a list from the leaf of a hash containing hashes that contain arrays
my @trans_verbs = %dictionary<verb><transitive><>;
```

Object-oriented programming

Perl supports object-oriented programming via a mechanism known as *blessing*. Any reference can be blessed into being an object of a particular class. A blessed object can have methods invoked on it using the "arrow syntax" which will cause Perl to locate or "dispatch" an appropriate subroutine by name, and call it with the blessed variable as its first argument.

While extremely powerful, it makes the most common case of object orientation, a struct-like object with some associated code, unnecessarily difficult. In addition, because Perl can make no assumptions about the object model in use, method invocation cannot be optimized very well.

In the spirit of making the "easy things easy and hard things possible", Raku retains the blessing model and supplies a more robust object model for the common cases.^[35] For example, a class to encapsulate a Cartesian point could be defined and used this way:

```
class Point is rw {
    has $.x;
    has $.y;

    method distance( Point $p ) {
        sqrt(($!x - $p.x) ** 2 + ($!y - $p.y) ** 2)
    }

    method distance-to-center {
        self.distance: Point.new(x=> 0, y => 0)
    }
}

my $point = Point.new( x => 1.2, y => -3.7 );
say "Point's location: (", $point.x, ', ', $point.y, ')';
# OUTPUT: Point's location: (1.2, -3.7)

# Changing x and y (note methods "x" and "y" used as lvalues):
$point.x = 3;
$point.y = 4;
say "Point's location: (", $point.x, ', ', $point.y, ')';
# OUTPUT: Point's location: (3, 4)

my $other-point = Point.new(x => -5, y => 10);
$point.distance($other-point); #=> 10
$point.distance-to-center;      #=> 5
```

The dot replaces the arrow in a nod to the many other languages (e.g. C++, Java, Python, etc.) that have coalesced around dot as the syntax for method invocation.

In the terminology of Raku, `$.x` is called an "attribute". Some languages call these *fields* or *members*. The method used to access an attribute is called an "accessor". An auto-accessor method is a method created automatically and named after the attribute's name, as the method `x` is in the example above. These accessor functions return the value of the attribute. When a class or individual attribute is declared with the `is rw` modifier (short for "read/write"), the auto-accessors can be passed a new value to set the attribute to, or it can be directly assigned to as an lvalue (as in the example). Auto-accessors can be replaced by user-defined methods, should the programmer desire a richer interface to an attribute. Attributes can only be accessed directly from within a class definition via the `$!` syntax regardless of how the attributes are declared. All other access must go through the accessor methods.

The Raku object system has inspired the Moose framework that introduces many of Raku's OOP features to Perl.

Inheritance, Roles and Classes

Inheritance is the technique by which an object or type can re-use code or definitions from existing objects or types. For example, a programmer may want to have a standard type but with an extra attribute. Inheritance in other languages, such as Java, is provided by allowing Classes to be sub-classes of existing classes.

Raku provides for inheritance via Classes, which are similar to Classes in other languages, and Roles.

Roles in Raku take on the function of *interfaces* in Java, *mixins* in Ruby, and *traits*^[36] in the Smalltalk variant Squeak. These are much like classes, but they provide a safer composition mechanism.^[37] These are used to perform composition when used with classes rather than adding to their inheritance chain. Roles define

nominal types; they provide semantic names for collections of behavior and state. The fundamental difference between a role and a class is that classes can be instantiated; roles are not.^[38]

Although Roles are distinct from Classes, it is possible to write Raku code that directly instantiates a Role or uses a Role as a type object, Raku will automatically create a class with the same name as the role, making it possible to transparently use a role as if it were a class.^[39]

Essentially, a role is a bundle of (possibly abstract) methods and attributes that can be added to a class without using inheritance. A role can even be added to an individual object; in this case, Raku will create an anonymous subclass, add the role to the subclass, and change the object's class to the anonymous subclass.

For example, a Dog is a Mammal because dogs inherit certain characteristics from Mammals, such as mammary glands and (through Mammal's parent, Vertebrate) a backbone. On the other hand, dogs also may have one of several distinct types of behavior, and these behaviours may change over time. For example, a Dog may be a Pet, a Stray (an abandoned pet will acquire behaviours to survive not associated with a pet), or a Guide for the blind (guide dogs are trained, so they do not start life as guide dogs). However, these are sets of additional behaviors that can be added to a Dog. It is also possible to describe these behaviors in such a way that they can be usefully applied to other animals, for example, a Cat can equally be a Pet or Stray. Hence, Dog and Cat are both distinct from each other, while both remaining within the more general category Mammal. So Mammal is a Class and Dog and Cat are classes that inherit from Mammal. But the behaviours associated with Pet, Stray, and Guide are Roles that can be added to Classes, or objects instantiated from Classes.

```
class Mammal is Vertebrate {  
    ...  
}  
class Dog is Mammal {  
    ...  
}  
role Pet {  
    ...  
}  
role Stray {  
    ...  
}  
role Guide {  
    ...  
}
```

Roles are added to a class or object with the `does` keyword. In order to show inheritance from a class, there is a different keyword `is`. The keywords reflect the differing meanings of the two features: role composition gives a class the *behavior* of the role, but doesn't indicate that it is truly the *same thing* as the role.

```
class GuideDog is Dog does Guide {  
    ...  
} # Subclass composes role  
  
my $dog = new Dog;  
$dog does Guide; # Individual object composes role
```

Although roles are distinct from classes, both are types, so a role can appear in a variable declaration where one would normally put a class. For example, a Blind role for a Human could include an attribute of type Guide; this attribute could contain a Guide Dog, a Guide Horse, a Guide Human, or even a Guide Machine.

```
class Human {  
    has Dog $dog; # Can contain any kind of Dog, whether it does the  
    ... # Guide role or not  
}
```



```

role Blind {
    has Guide $guide; # Can contain any object that does the Guide role,
    ...               # whether it is a Dog or something else
}

```

Regular expressions

Perl's regular expression and string-processing support has always been one of its defining features.^[40] Since Perl's pattern-matching constructs have exceeded the capabilities of regular language expressions for some time,^[41] Raku documentation will exclusively refer to them as *regexes*, distancing the term from the formal definition.

Raku provides a superset of Perl features with respect to regexes, folding them into a larger framework called "rules" which provide the capabilities of context-sensitive parsing formalisms (such as the syntactic predicates of parsing expression grammars and ANTLR), as well as acting as a closure with respect to their lexical scope.^[42] Rules are introduced with the `rule` keyword which has a usage quite similar to subroutine definition. Anonymous rules can also be introduced with the `regex` (or `rx`) keyword, or they can simply be used inline as regexps were in Perl via the `m` (matching) or `s` (substitute) operators.

In *Apocalypse 5*, Larry Wall enumerated 20 problems with "current regex culture". Among these were that Perl's regexes were "too compact and 'cute'", had "too much reliance on too few metacharacters", "little support for named captures", "little support for grammars", and "poor integration with 'real' language".^[43]

Syntactic simplification

Some Perl constructs have been changed in Raku, optimized for different syntactic cues for the most common cases. For example, the parentheses (round brackets) required in control flow constructs in Perl are now optional:^[34]

```

if is_true() {
    for @array {
        ...
    }
}

```

Also, the `,` (comma) operator is now a list constructor, so enclosing parentheses are no longer required around lists. The code

```
@array = 1, 2, 3, 4;
```

now makes `@array` an array with exactly the elements '1', '2', '3', and '4'.

Chained comparisons

Raku allows comparisons to "chain". That is, a sequence of comparisons such as the following is allowed:

```

if 20 <= $temperature <= 25 {
    say "Room temperature is between 20 and 25!"
}

```

This is treated as if each left-to-right comparison were performed on its own, and the result is logically combined via the `and` operation.

Lazy evaluation

Raku uses the technique of lazy evaluation of lists that has been a feature of some functional programming languages such as Haskell.^[44]

```
@integers = 0..Inf; # integers from 0 to infinity
```

The code above will not crash by attempting to assign a list of infinite size to the array `@integers`, nor will it hang indefinitely in attempting to expand the list if a limited number of slots are searched.

This simplifies many common tasks in Raku including input/output operations, list transformations, and parameter passing.

Gather

Related to lazy evaluation is the construction of lazy lists using `gather` and `take`, behaving somewhat like generators in languages like Icon or Python.

```
my $squares = lazy gather for 0..Inf {  
    take $_ * $_;  
};
```

`$squares` will be an infinite list of square numbers, but lazy evaluation of the `gather` ensures that elements are only computed when they are accessed.

Junctions

Raku introduces the concept of *junctions*: values that are composites of other values.^[44] In their simplest form, junctions are created by combining a set of values with junctive operators:

```
# Example for | ("any") Junction:  
my $color = 'white';  
unless $color eq 'white' | 'black' | 'gray' | 'grey' {  
    die "Color printing not supported\n";  
}  
  
# Example for & ("all") Junction:  
my $password = 'secret!123';  
if $password ~~ /<:alpha>/ & /<:digit>/ & /<:punct>/ {  
    say "Your password is reasonably secure";  
}
```

`|` indicates a value which is equal to either its left- or right-hand arguments. `&` indicates a value which is equal to both its left- *and* right-hand arguments. These values can be used in any code that would use a normal value. Operations performed on a junction act on all members of the junction equally, and combine according to the junctive operator. So, `("apple"|"banana") ~ "s"` would yield `"apples"|"bananas"`.

In comparisons, junctions return a single true or false result for the comparison. "any" junctions return true if the comparison is true for any one of the elements of the junction. "all" junctions return true if the comparison is true for all of the elements of the junction.

Junctions can also be used to more richly augment the type system by introducing a style of generic programming that is constrained to junctions of types:

```
subset Color of Any where RGB_Color | CMYK_Color;
sub get_tint(Color $color, Num $opacity) {
    ...
}
```

Macros

In low-level languages, the concept of macros has become synonymous with textual substitution of source-code due to the widespread use of the C preprocessor. However, high-level languages such as Lisp pre-dated C in their use of macros that were far more powerful.^[45] It is this Lisp-like macro concept that Raku will take advantage of.^[33] The power of this sort of macro stems from the fact that it operates on the program as a high-level data structure, rather than as simple text, and has the full capabilities of the programming language at its disposal.

A Raku macro definition will look like a subroutine or method definition, and it can operate on unparsed strings, an AST representing pre-parsed code, or a combination of the two. A macro definition would look like this:^[46]

```
macro hello($what) {
    quasi { say "Hello { {{{$what}}} }" };
}
```

In this particular example, the macro is no more complex than a C-style textual substitution, but because parsing of the macro parameter occurs before the macro operates on the calling code, diagnostic messages would be far more informative. However, because the body of a macro is executed at compile time each time it is used, many techniques of optimization can be employed. It is even possible to entirely eliminate complex computations from resulting programs by performing the work at compile-time.

Identifiers

In Perl, identifier names can use the ASCII alphanumerics and underscores also available in other languages. In Raku, the alphanumerics can include most Unicode characters. In addition, hyphens and apostrophes can be used (with certain restrictions, such as not being followed by a digit). Using hyphens instead of underscores to separate words in a name leads to a style of naming called "kebab case".

Examples

Hello world

The hello world program is a common program used to introduce a language. In Raku, hello world is:

```
say 'Hello, world';
```

— though there is more than one way to do it.

Factorial

The factorial function in Raku, defined in a few different ways:

```
# Using recursion (with `if\else` construct)
sub fact( UInt $n --> UInt ) {
    if $n == 0 { 1 }
    else      { $n * fact($n-1) }
}

# Using recursion (with `if` as statement modifier)
sub fact( UInt $n --> UInt ) {
    return 1 if $n == 0;
    return $n * fact($n-1);
}

# Using recursion (with `when` construct)
sub fact( UInt $n --> UInt ) {
    when $n == 0 { 1 }
    default      { $n * fact($n-1) }
}

# Using the ternary operator
sub fact( UInt $n --> UInt ) {
    $n == 0 ?? 1 !! $n * fact($n-1)
}

# Using multiple dispatch
multi fact(0) { 1 }
multi fact( UInt $n --> UInt ) {
    $n * fact($n - 1)
}

# Using the reduction metaoperator
multi fact( UInt $n --> UInt ) {
    [*] 1..$n
}

# Creating a factorial operator and using the reduction metaoperator
sub postfix:<!>( UInt $n --> UInt ) { [*] 1..$n }

# Using `state` declarator to create a memoized factorial
sub fact( UInt $n --> UInt ) {
    state %known = 0 => 1;
    return %known{$n} if %known{$n}:exists;
    %known{$n} = $n * fact($n-1);
    return %known{$n};
}
```

Quicksort

Quicksort is a well-known sorting algorithm. A working implementation using the functional programming paradigm can be succinctly written in Raku:

```
# Empty list sorts to the empty list
multi quicksort([]) { () }

# Otherwise, extract first item as pivot...
multi quicksort([$pivot, *@rest]) {
    # Partition.
    my @before = @rest.grep(* before $pivot);
    my @after  = @rest.grep(* after $pivot);
```

```

# Sort the partitions.
flat (quicksort(@before), $pivot, quicksort(@after))
}

```

Tower of Hanoi

Tower of Hanoi is often used to introduce recursive programming in computer science. This implementation uses Raku's multi-dispatch mechanism and parametric constraints:

```

multi sub hanoi(0, $, $, $) { } # No disk, so do not do anything
multi sub hanoi($n, $a = 'A', $b = 'B', $c = 'C') { # Start with $n disks and three pegs A,
  B, C # firstly move top $n - 1 disks from A
  hanoi $n - 1, $a, $c, $b; # then move last disk from A to C
  to B # lastly move $n - 1 disks from B to C
  say "Move disk $n from peg $a to peg $c";
  hanoi $n - 1, $b, $a, $c;
}

```

Books

In the history of Raku there were two waves of book writing. The first wave followed the initial announcement of Perl 6 in 2000. Those books reflect the state of the design of the language of that time, and contain mostly outdated material. The second wave, that followed the announcement of Version 1.0 in 2015, includes several books that have already been published and some others that are in the process of being written.

Books published before Perl 6 version 1.0 (known as version 6.c)

- A. Randal, D. Sugalski, L. Totsch. *Perl 6 and Parrot Essentials*, 1st edition, 2003, [ISBN 978-0596004996](#)
- A. Randal, D. Sugalski, L. Totsch. *Perl 6 and Parrot Essentials*, Second Edition (<http://shop.oreilly.com/product/9780596007379.do>) 2004. [ISBN 978-0596007379](#)
- S. Walters. *Perl 6 Now: The Core Ideas Illustrated with Perl 5*. (<http://www.apress.com/us/book/9781590593950>) 2004. [ISBN 978-1590593950](#)

Also, a book dedicated to one of the first Perl 6 virtual machines, Parrot, was published in 2009.

- A. Randal, A. Whitworth. *Parrot Developer's Guide: Pir*. (<http://onyxneon.com/books/pir/index.html>) 2009. [ISBN 978-0977920129](#)

Books published after Perl 6 version 1.0 (known as version 6.c)

- Andrew Shitov. *Perl 6 at a Glance*. (<https://andrewshitov.com/perl6-at-a-glance/>) 2017. ISBN 978-90-821568-3-6 — download free official PDF (<https://andrewshitov.com/wp-content/uploads/2020/01/Perl-6-at-a-Glance.pdf>) or download it from GitHub (<https://github.com/ash/book/blob/master/Perl%206%20at%20a%20Glance/Perl%206%20at%20a%20Glance.pdf>)
- Laurent Rosenfeld, Allen B. Downey. *Think Perl 6* (<http://shop.oreilly.com/product/0636920065883.do>) 2017. [ISBN 978-1-4919-8055-2](#)
- Moritz Lenz. *Perl 6 Fundamentals* (<https://www.apress.com/us/book/9781484228982>) 2017. [ISBN 978-1-4842-2898-2](#)

- J. J. Merelo. *Learning to program with Perl 6: First Steps: Getting into programming without leaving the command line* (<https://github.com/JJ/perl6em/tree/master/docs>) 2017. ISBN 978-1521795781
- Andrew Shitov. *Perl 6 Deep Dive* (<https://www.packtpub.com/application-development/perl-6-deep-dive>). 2017. ISBN 978-1-78728-204-9
- Andrew Shitov. *Using Perl 6* (<https://andrewshitov.com/using-perl6/>). 2017. ISBN 978-90-821568-1-2 — download free official PDF (<https://andrewshitov.com/wp-content/uploads/2020/01/Using-Perl-6.pdf>) or download from GitHub (<https://github.com/ash/books/blob/master/Using%20Perl%206/Using%20Perl%206.pdf>)
- Moritz Lenz. *Searching and Parsing with Perl 6 Regexes* (<https://www.apress.com/us/book/9781484232279>) 2017. ISBN 978-1-4842-3228-6
- brian d foy. *Learning Perl 6* (<http://shop.oreilly.com/product/0636920062776.do>) 2018 ISBN 978-1491977682

Books published with the new Raku name

- Andrew Shitov. *Using Raku* (<https://andrewshitov.com/2019/10/13/using-raku-the-free-book/>), 2nd edition, 2019, ISBN 978-90-821568-8-1 — download free official PDF (<https://andrewshitov.com/wp-content/uploads/2020/01/Using-Raku.pdf>) or download a copy from GitHub (<https://github.com/ash/books/blob/master/Using%20Raku/Using%20Raku.pdf>)
- Andrew Shitov. *Raku One-Liners* (<https://andrewshitov.com/2019/10/18/raku-one-liners-a-free-book/>), 2019, ISBN 978-90-821568-9-8 — download free official PDF (<https://andrewshitov.com/wp-content/uploads/2020/01/Raku-One-Liners.pdf>) or download from GitHub (<https://github.com/ash/books/blob/master/Raku%20One-Liners/Raku%20One%20Liners.pdf>)

Books to be published

There are a few reports^[47] from different authors about the new books that are going to be published soon, all based on the current version 1.0 (known as version 6.c) of Perl 6.

- Andrew Shitov. *Migrating to Perl 6. An Easy Transition from Perl 5*. ISBN 978-90-821568-5-0. Probably never to be published.
- Gabor Szabo. *Web Application Development in Perl 6* (<https://leanpub.com/bailador>).


References

1. Jackson, Joab (23 July 2010). "Perl creator hints at imminent release of long-awaited Perl 6" (<http://www.infoworld.com/d/open-source/perl-creator-hints-imminent-release-long-awaited-perl-6-445>). IDG News Service. Retrieved 8 February 2015.
2. http://blogs.perl.org/users/zoffix_znet/2018/11/announce-raku-perl-6-diwali-6d-language-specification-release.html
3. https://docs.raku.org/language/modules#Basic_structure
4. "Glossary of Terms and Jargon" (https://web.archive.org/web/20120121145808/http://www.perlfoundation.org/perl6/index.cgi?glossary_of_terms_and_jargon). *Perl Foundation Perl 6 Wiki*. The Perl Foundation. 28 February 2011. Archived from the original (http://www.perlfoundation.org/perl6/index.cgi?glossary_of_terms_and_jargon) on 21 January 2012. Retrieved 8 February 2015.
5. 唐鳳, a.k.a. Audrey Tang (21 April 2010). "How to Implement Perl 6 in '10" (<http://pugs.blogs.com/pugs/2010/04/how-to-implement-perl-6-in-10.html>).

6. "About Perl" (<http://www.perl.org/about.html>). perl.org. Retrieved 11 June 2020. "'Perl' is a family of languages, 'Raku' (formerly known as 'Perl 6') is part of the family, but it is a separate language which has its own development team. Its existence has no significant impact on the continuing development of 'Perl'."
7. "Perl 6 renamed to Raku" (<https://lwn.net/Articles/802329/>). LWN.net. 15 October 2019. Retrieved 16 October 2019.
8. "TPF response to Raku rename" (<https://news.perlfoundation.org/post/tpf-response-raku-rename>). 29 October 2019.
9. Federico Biancuzzi; Shane Warden. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. ISBN 978-0596515171.
10. Kline, Joe (21 August 2000). "Report from the Perl Conference" (<http://www.perl.com/pub/a/2000/08/tpc4.html>).
11. Wall, Larry (2000). "State of the Onion 2000" (<http://www.perl.com/pub/a/2000/10/23/soto2000.html>). O'Reilly Network.
12. The Perl Foundation (2000). "About Perl 6 RFCs" (<https://raku.org/archive/rfc/meta/>).
13. Wall, Larry (2 April 2001). "Apocalypse 1: The Ugly, the Bad, and the Good" (<https://raku.org/archive/doc/design/apo/A01.html>).
14. "Raku test suite" (<https://github.com/Raku/roast>). 2019.
15. Larry Wall and the Perl 6 designers (2015). "Perl 6 Design Documents" (<https://design.raku.org/>).
16. The Perl Foundation (2001). "Exegeses" (<https://raku.org/archive/doc/exegesis.html>).
17. The Perl Foundation (2002). "Perl Development: Mailing Lists" (<https://raku.org/archive/lists/>).
18. "Larry Wall in IRC chat log" (https://irclog.perlgeek.de/perl6/2016-01-15#i_11894111). 15 January 2016. Retrieved 10 November 2017.
19. "Archived 'Logo considerations' email from Larry Wall" (<https://github.com/perl6/mu/blob/aaa173d49c8ba681628739e96ed28fdb921211ec/misc/camelia.txt#L78-L80>). 24 March 2009. Retrieved 10 November 2017.
20. Wall, Larry (10 August 2004). "Synopsis 1: Overview" (<https://design.raku.org/S01.html>).
21. "rakudo/rakudo - GitHub" (<https://github.com/rakudo/rakudo/>). Github.com. Retrieved 21 September 2013.
22. Michaud, Patrick (16 January 2008). "The compiler formerly known as 'perl6'" (<https://web.archive.org/web/20120218080103/http://use.perl.org/~pmichaud/journal/35400>). Archived from the original (<http://use.perl.org/~pmichaud/journal/35400>) on 18 February 2012.
23. Worthington, Jonathan. "MoarVM: A virtual machine for NQP and Rakudo" (<http://6guts.wordpress.com/2013/05/31/moarvm-a-virtual-machine-for-nqp-and-rakudo/>). 6guts. Retrieved 24 July 2013.
24. "MoarVM" (<http://moarvm.com/>). MoarVM team. Retrieved 8 July 2017.
25. "Feature comparison of Perl 6 compilers" (<https://web.archive.org/web/20190207035658/http://perl6.org/compilers/features>). Archived from the original (<http://perl6.org/compilers/features>) on 7 February 2019.
26. Wall, Larry; et al. (2007). "Perl 6 STD" (<https://github.com/perl6/std/blob/master/STD.pm6>).
27. "mp6/kp6 FAQ" (<http://darcs.pugscode.org/v6/v6-KindaPerl6/docs/FAQ.pod>). Perl 6 development team. 2006.
28. "Yapsi README" (<https://github.com/masak/yapsi/blob/master/README>). 2011.
29. O'Rear, Stefan (29 November 2011). "Niecza README.pod" (<https://github.com/sorear/niecza/blob/master/README.pod>). Retrieved 12 January 2012.
30. Wall, Larry (2004). "Synopsis 11: Modules" (<https://design.raku.org/S11.html>).
31. "Perl 6 Modules Directory" (<https://modules.raku.org/>). Modules.raku.org. Retrieved 17 May 2020.

32. Wall, Larry (20 May 2009). "Synopsis 2: Bits and Pieces" (<https://design.raku.org/S02.html>).
33. Wall, Larry (21 March 2003). "Synopsis 6: Subroutines" (<https://design.raku.org/S06.html>).
34. Wall, Larry (20 May 2009). "Synopsis 4: Blocks and Statements" (<https://design.raku.org/S04.html>).
35. Wall, Larry (18 August 2006). "Synopsis 12: Objects" (<https://design.raku.org/S12.html>).
36. The Software Composition Group (2003). "Traits" (<https://web.archive.org/web/20060811170712/http://www.iam.unibe.ch/~scg/Research/Traits/>). Archived from the original (<http://www.iam.unibe.ch/~scg/Research/Traits/>) on 11 August 2006. Retrieved 22 September 2006.
37. Jonathan Worthington (2009). "Day 18: Roles" (<https://perl6advent.wordpress.com/2009/12/18/day-18-roles/>).
38. chromatic (2009). "The Why of Perl Roles" (<http://www.modernperlbooks.com/mt/2009/04/the-why-of-perl-roles.html>).
39. "Object orientation" (https://docs.raku.org/language/objects#Automatic_role_punning). *docs.raku.org*. Retrieved 24 October 2019.
40. Parlante, Nick (2000). "Essential Perl: String Processing with Regular Expressions" (<http://cslibrary.stanford.edu/108/EssentialPerl.html#re>).
41. Christiansen, Tom (1996). "PERL5 Regular Expression Description" (<https://web.archive.org/web/20100331122815/http://www.perl.com/doc/FMTEYEWTK/regexps.html>). Archived from the original (<http://www.perl.com/doc/FMTEYEWTK/regexps.html>) on 31 March 2010. Retrieved 25 March 2010. "Perl's regexps "aren't" -- that is, they aren't "regular" because backreferences per sed and grep are also supported, which renders the language no longer strictly regular"
42. Wall, Larry (20 May 2009). "Synopsis 5: Regexes and Rules" (<https://design.raku.org/S05.html>).
43. Wall, Larry (4 June 2002). "Apocalypse 5: Pattern Matching" (<https://raku.org/archive/doc/design/apo/A05.html>).
44. Wall, Larry (13 September 2004). "Synopsis 9: Data Structures" (<https://design.raku.org/S09.html>).
45. Lamkins, David B. (8 December 2004). *Successful Lisp: How to Understand and Use Common Lisp* (<http://psg.com/~dlamkins/sl/chapter20.html>). bookfix.com.
46. "Macros" (<https://design.raku.org/S06.html#Macros>).
47. Books about Perl 6 (<https://perl6book.com/>)

External links

- [Official website \(https://raku.org/\)](https://raku.org/) 

Retrieved from "[https://en.wikipedia.org/w/index.php?title=Raku_\(programming_language\)&oldid=985273096](https://en.wikipedia.org/w/index.php?title=Raku_(programming_language)&oldid=985273096)"

This page was last edited on 25 October 2020, at 00:42 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.