Karel Language Reference Manual

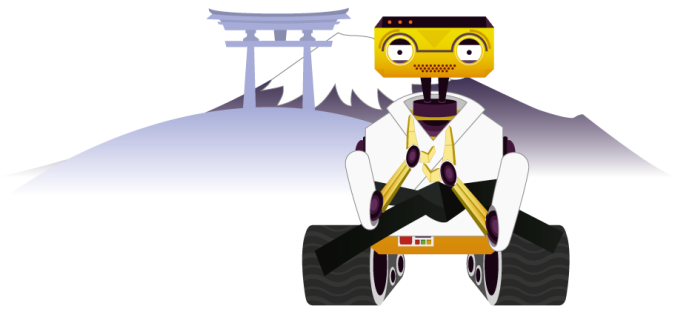
August 23, 2016[1]

Contents

[1] This document was prepared using the LaTeX module in NCLab

1 About this document

This document covers selected basic functionality of Karel the Robot, and it is meant to be a reference rather than a learning material. If you would like to learn computer programming with Karel, take the NCLab's self-paced Karel Programming Course. More about the course can be found at `https://nclab.com/karel/`.



2 Language variants

Karel can be used with English, Spanish, German, Czech, Polish, Italian, and French commands. All language versions of this document can be found in the menu of the Karel module. Any of these seven languages can also be chosen in Settings to be the main language for NCLab.

3 Karel module in NCLab

The Karel module in NCLab has four modes:

- Manual mode: Guide the robot using the mouse or keyboard
- Programming: Write and run programs.
- Designer: Design your own mazes.
- Games: Design your own games.

You can publish all your programs and games on the web and on social networks using the function "Publish to the web".

4 Basic commands

Karel knows five basic commands that are equivalent to the five buttons in manual mode:

- `go`: Make one step forward.

- `left`: Turn 90 degrees left.
- `right`: Turn 90 degrees right.
- `get`: Collect an object from the ground.
- `put`: Put an object on the ground.

5 Conditions

The `if-elif-else` conditions help the robot check his surroundings and make decisions at runtime. Notice the indentation. Every condition may have zero or more branches `elif` and zero or one branch `else`. Example:

```
if wall
  left
elif stone
  right
elif rock
  right
  right
else
  go
```

6 Loops

There are three types of loops: `repeat`, `while`, and `for-in`. The `repeat` loop is used when the number of repetitions is known in advance:

```
repeat 5
  go
```

The `while` loop should be used when we do not know in advance how many repetitions will be needed:

```
while not wall
  go
```

If we want to iterate over every element of a given sequence, it is convenient to use the loop `for-in`. After the keyword `for` we write a name of a variable that will store the value of one element of the given sequence (one iteration – one value). After the keyword `in` we write a sequence, i.e., a list, a string, or a set of expressions, separated by the comma `,`. Examples:

```
for x in [0, 1, 2, 3]
  print x
```

```
for y in "Hello, World!"
  print y

for z in 100, "Hi!", [10, 20]
  print z
```

Every loop can contain basic commands, conditions, other loops, and custom commands. Also, every loop may have the branch `else`. For example:

```
while wall
  left
else
  go
```

7 Custom commands

Custom commands should be defined for "smaller tasks" that are done more than once in the program:

```
def turnback
  repeat 2
    left
```

Every custom command may have a set of local variables which are created during the time of registration of a custom command. These local variables should be listed after the name of a command and are usually called *parameters*. For example, the following command has one parameter `x`:

```
def goForward(x)
  repeat x
    go
```

If we want to go `5` steps forward, we call the command `goForward` as follows:

```
goForward(5)
```

In this case, the number `5` is called an *argument*. Parameters of any command may have their own default values:

```
def goForwardMore(x = 3)
  repeat x
    go
```

If a command has a parameter with a default value, we may omit the corresponding argument. For example, after the following call, Karel the Robot will be `3` steps ahead:

```
goForwardMore
```

8 Logical expressions

Keyword `not` means *negation*. It returns `True` if the operand is `False` and vice versa. Example:

```
while not wall
  go
```

Keyword `and` returns `True` if both statements are `True`, else it returns `False`. Example:

```
while not wall and not home
  go
```

Keyword `or` returns `True` if at least one of the statements is `True`, otherwise it returns `False`. Example:

```
if gem or nugget
  get
```

9 GPS coordinates

Karel can retrieve his GPS coordinates using the commands `gpsx` and `gpsy`. In the Southwest corner of the maze, both `gpsx` and `gpsy` are `0`. In the Northeast corner, `gpsx` is `14` and `gpsy` is `11`. Both GPS coordinates can be used together with variables – see Section 11.

10 Randomness

The command `rand` returns randomly `True` or `False`. This allows Karel to make random moves. Example:

```
if rand
  left
else
  right
```

Also, Karel has the function `randint` that returns a random *integer* number from the interval `[1, n]`, if the following call is used:

```
randint(n)
```

If we call

```
randint(m, n)
```

a random number from the interval `[m, n]` is returned. Here, `m` and `n` may be any integer numbers. For example, the following call returns a random integer number from the interval `[10, 100]`:

```
randint(10, 100)
```

11 Variables

Variables are used to store information. Names of Karel variables are case-sensitive, i.e., the letter case matters. For example, the following two names are different: `n` and `N`. Names can contain letters from various alphabets. For example, from English, Greek, Czech, Polish, Russian, etc. alphabets. Karel knows three types of variables:

1. *Numerical variables* store integer and real numbers:

   ```
   n1 = gpsx
   n2 = gpsy
   n3 = 15
   n4 = 1000.5
   ```

   They can be increased by one via the command `inc` and decreased by one via the command `dec`:

   ```
   inc(n1)
   dec(n2)
   ```

   They can also be increased or decreased by more than one:

   ```
   inc(n1, 3)
   dec(n2, 2)
   ```

   We can perfom the basic mathematical operations with numbers: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulus), `**` (exponentiation). For example:

   ```
   n = 10
   n + 12 - n / 2
   ```

   Also, Karel allows us to use the following augmented assignment statements: `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, i.e, instead of writing:

   ```
   a = 2
   a = a + 5
   a = a * 10
   ```

   we can write:

   ```
   a = 2
   a += 5
   a *= 10
   ```

2. *Text variables* store text strings:

```
my_name = "Karel"
my_surname = '...'
```

Karel can concatenate two or more strings using the operator `+`:

```
print 'Hello' + ", World" + '!' # Hello, World!
```

Also, every string can be "multiplied" by the given number:

```
'Hi' * 3
5 * "a"
```

The first expression returns the string `HiHiHi`. The second one – the string `aaaaa`. If we want to check, if a certain substring belongs to another string, we may use the operator `in`. The following expression returns the logical value `True`:

```
'aB' in 'AAaBBb'
```

3. *Logical variables* store logical values (`True`, `False`):

```
karel_is_west = (gpsx == 0)
karel_faces_north = north
```

Variables can be printed using the command `print`:

```
print "Value of n1 is", n1
```

or the function `print()`:

```
print("Karel ", my_surname)
```

Every Karel code can contain either the statement(s) `print` or the function(s) `print()`. Both cases are not possible. Also, variables can be deleted using the statement `del`:

```
a = 14
b = 'Hello'
del a, b
```

12 Lists

Karel provides basic functionality of Python lists. As in Python, indices start with zero and they can be negative:

- `L = []` ... creates an empty list `L`.
- `L = L1 + L2` ... concatenates two lists `L1` and `L2`.
- `L = L * n` or `L = n * L` ... multiplies the list `L1` `n` times.

- `len(L)` ... returns the length of the list `L`.
- `L[i]` ... returns item at position `i`.
- `L.append(x)` ... appends item `x` at the end of `L`.
- `y = L.pop()` ... removes and returns the last item from `L`.
- `y = L.pop(i)` ... removes and returns item at position `i`.
- `y = L.pop(0)` ... removes and returns the first item of `L`.
- `expr in L` ... returns `True`, if the `expr` belongs to the list `L`; otherwise, returns `False`.
- `del L[i]` ... removes item at position `i`.

13 Functions

Functions in Karel are similar to custom commands, but they can return values via the command `return`. Example:

```
def measure_distance
  n = 0
  while not wall
    inc(n)
    go
  return n
```

Custom functions may also have parameters and parameters with default values:

```
def sum(a, b, c = 10)
  return a + b + c
```

14 Recursion

Karel is capable of recursion, which means that a custom command or function can call itself. Example:

```
def climb_stairs
  while wall
    left
    go
    right
    go
    climb_stairs
```

Commands or functions can also be mutually recursive (function A calls function B and at the same time function B calls function A).

15 The statement `pass`

Karel has the statement `pass` that does nothing. This statement is useful in several situations. For example, if you want to iterate over every element of the list `[[], [0], ['Hello'], [True, False]]`, but you still did not decide what to do with the empty list `[]`, you can use the statement `pass`:

```
for x in [[], [0], ['Hello'], [True, False]]
  if x == []
    pass
  else
    print x
```