porcupy Documentation

Release 0.1

Sviatoslav Abakumov

Oct 26, 2017

Contents

Welcome to Porcupy's documentation. *Porcupy* let's you compile a subset of Python to Yozhiks in Quake II scenarios.

Check *Introduction* to find out about *Yozhiks in Quake II* and its scenarios. Go to *Quickstart* to learn how to install and write your first scenario in Porcupy.

CHAPTER 1

Introduction

Yozhiks in Quake II is video game developed by Nick Koleda between 2001-2006. It's a 2D side-scrolling demake of *Quake II* where player controls a *yozhik* (hedgehog) battling other yozhiks. It features single player with bots, hot seat, LAN and Internet multiplayer games with following game modes: Deathmatch, Team Deathmatch, Capture the Flag, Points, Scenario. There is a map editor to create and edit maps.

The game gained a short-lived fan following in Russia and influenced *3d[Power]* to create NFK.

More interestingly it features a tiny custom-made programming language to write *scenarios* attached to a map. With this language, map designer is able to create scripted events, spawn and control bots, access and modify the state of game objects like doors, buttons, and other players.

However, the code is pretty arcane:

```
# e1p < 1 ( p1z ~5  p1z p1z+1  e1b ^1  e1p 125 )
```

It reads like this: if first yozhik has less health points than 1, then spawn him in a random spawn-point between 1 and 5, and give him 125 health points.

As you can see, reading and writing such code is not an easy feat: arbitrary one-letter abbreviations of methods and properties, unnamed variables, flow control consists of one-level `if` statement and `goto` statement.

Head over to *Quickstart* and learn how to install and write scenarios in Porcupy.

CHAPTER 2

Quickstart

Installation

First of all, you must visit Yozhiks in Quake II and download the game and map editor.

Make sure you have Python 3, then use *pip* to acquire the package:

```
pip3 install 'git+https://github.com/Perlence/porcupy#egg=porcupy'
```

First scenario

Let's consider rewriting an *example* from *Introduction* in Porcupy:

```
PLAYER = yozhiks[0]

if PLAYER.health < 1:
    PLAYER.spawn(randint(1, 5))
    PLAYER.health = 125
```

List *yozhiks* is a zero-indexed list of all yozhiks in the game and each of them has attributes like *health* and *weapon*, and methods like *spawn()*.

Names written in upper case are considered constants, so, roughly speaking, each next occurrence of `PLAYER` after first line will be replaced by `yozhiks[0]`.

New built-in function *randint()* returns random integer in range `[a, b]`, including both end points.

Let's save the scenario in a file, e.g. `handicap_spawn.py`, and see what Porcupy compiler will produce:

```
porcupy -i handicap_spawn.py
```

The result is:

```
# e1p >= 1 ( g1z ) p1z ~5 p2z p1z+1 e1b ^2 e1p 125 :1
```

It looks a lot like the original example, but you can notice new words like `g1z` and `:1` — these are *goto* statement and *goto* label respectively.

Now, the scenario on it's own is useless unless it's bundled with a map. Go to directory where you installed Yozhiks in Quake II and the map editor. Start `red_egiks.exe`, open file `MAPS/ArenaDM.egm`, change the name of map in *Info* dialog, and save it to `MAPS/ArenaDM2.egm`.

Now we can compile and attach the Porcupy scenario to the map:

```
porcupy -i handicap_spawn.py -a ArenaDM2.egm
```

Check if scenario works properly by loading the map in Yozhiks in Quake II and proceed to *Reference*.

CHAPTER 3

Reference

Porcupy compiler uses Python's ast module to parse Porcupy scenarios. Porcupy aims to resemble Python as close as possible, with some cues taken from Go. But it's not feasible to implement each and every one of Python language features.

Here's a list of Python language features not supported in Porcupy:

- The import system
- Expressions:
  - Await expression
  - Power operator
  - Shifting operations
  - Binary bitwise operations
  - Lambdas
  - Keywords in function calls
  - `list`, `set`, `dict`, and generator comprehensions
- Simple statements:
  - The `assert` statement
  - The `del` statement
  - The `return` statement
  - The `yield` statement
  - The `raise` statement
  - The `import` statement
  - The `global` statement
  - The `nonlocal` statement

- Compound statements:
  - The `try` statement
  - The `with` statement
  - Function definitions
  - Class definitions
  - Coroutines

Identifiers and assignment

Unlike Python, Porcupy introduces a distinction between variables and constants. Constants are not assigned to in-game variables, like `plz`, the value of constant is stored only in compiler's memory. To define a constant, write its name in upper case:

```
PUNCH_VELOCITY = 15
```

Note: Because of the way the game parses floating point numbers, and because of the way Porcupy tries to alleviate it, defining floating point constants is not allowed.

All other names are considered variable names:

```
number = 0
```

Chained assignment and tuple unpacking are supported:

```
x = y = 0
a, b = 1, 2
```

Note: Although present in game, string variables are broken, and it's not possible to set a string to a variable. At the same time, it's still possible to define a string constant.

Data types

Porcupy supports the following data types:

Numbers Integers

Booleans

Floating point numbers

Sequences All sequences provide a way to get/set an item by index, query the length and capacity.

Immutable sequences

Strings Can only be used in constants and as *print()*, *print_at()*, and *load_map()* arguments. Method `format` is supported:

```
print('{} {}'.format(yegiks[0].health, yegiks[0].armor)
```

Range See the *range* built-in.

Reversed See the *reversed* built-in.

Mutable sequences

Lists The items of a list are of the same type and the number of items is constant and known at compile-time:

```
x = [0, 1, 2, 3, 4]
```

No original list methods are implemented in Porcupy lists, it can only be used to store a sequence of numbers, get and set them by index:

```
x[0] = 10
print(x[0])
print(len(x))
```

Note: Negative indices are not supported.

Slices Slice is a variable-length sequence with defined maximum capacity, backed by a list. Essentially, slice is a triple of values: address of first element, length of slice, capacity of slice.

```
x = [0, 0, 0, 0, 0]  # a list of length 5
s = x[:]  # a slice of list *x*, length 5, capacity 5
s = x[1:]  # a slice of list *x*, length 4, capacity 4
s = x[:0]  # a slice of list *x*, length 0, capacity 5
s = x[1:3]  # a slice of list *x*, length 3, capacity 4
```

Note: Slice step is not supported.

There's a very useful shorthand notation with *slice()*.

It's possible to slice other slices:

```
x = slice(int, 5)
y = x[:3]
```

Slices can be appended to:

```
x = slice(int, 0, 5)
x.append(4)
```

Warning: There's currently no mechanism to prevent user from appending an item to a "full" slice, so be sure to check length and capacity of slice before appending yourself.

Compound statements

Only the following compound statements from Python are supported:

- The `if` statement

- The `while` statement
- The `for` statement

Each of them supports optional `else` clause.

The `for` statement differs a bit from the original. It can be used to iterate sequences:

```
items = [10, 20, 30, 40]
for item in items:
    print(item)  # prints '10', '20', '30', '40', one on each line
```

But it's also possible to access item's index without the `enumerate` function:

```
items = [10, 20, 30, 40]
for i, item in items:
   print(i, item)  # prints '0 10', '1 20', and so on
```

Built-in functions

cap (*sequence*) → int
Return the capacity of a given sequence.

Parameters **sequence** – an instance of list, slice, range, or reversed.

len (*sequence*) → int
Return the length of a given sequence.

Parameters **sequence** – a list, slice, range, or reversed.

load_map (*map_name*)
Load the given map.

Note: This function works only in Yozhiks in Quake II v1.07.

print (**values*)
Print *values* as a message in the top-left corner of the screen, separated by a single space.

print_at (*x*, *y*, *duration*, **values*)
Print *values* in given point on screen for *duration* game ticks, separated by a single space.

Parameters

- **x** (*int*) – *x* coordinate of message.
- **y** (*int*) – *y* coordinate of message.
- **duration** (*int*) – number of game ticks the message will be visible.
- **values** – parts of message to be printed.

Note: Only 20 such messages can be shown at a given time.

randint (*a*, *b*) → int
Return a random integer *N* such that `a <= N <= b`.

class range (*stop*) → range object

class range (*start*, *stop*[, *step*]) → range object
Return an object that produces a sequence of integers from start (inclusive) to stop (exclusive) by step.

class reversed (*sequence*) → reversed object
Return a reverse sequence without allocating any in-game variables.

set_color (*r*, *g*, *b*)
Set color of *print_at()* messages.

slice (*type*, *len*, *cap=None*) → slice object
Create a slice of capacity *cap* and *len* zero elements of given *type*.

Parameters

- **type** – int, bool, or float.
- **len** (*int*) – length of slice to make.
- **cap** (*int*) – capacity of slice to make, defaults to *len*.

```
x = slice(int, 5)  # equivalent to [0, 0, 0, 0, 0][:]
x = slice(int, 1, 5)  # equivalent to [0, 0, 0, 0, 0][:1]
y = slice(bool, 3)  # equivalent to [False, False, False][:]
z = slice(float, 5)  # equivalent to [.0, .0, .0, .0, .0][:]
```

spawn_sheep (*start*, *finish*)
Spawn a sheep in point *start* and tell it to go to point *finish*.

Parameters

- **start** (Point) – point where sheeps spawns.
- **finish** (Point) – point where sheep is supposed to go.

Note: Although point *finish* is required, only green sheeps will go there, other sheeps will always follow player.

Game objects

Porcupy provides access to many built-in objects to interact with the game.

bots
A list of 10 *Bot* instances.

buttons
A list of 50 *Button* instances.

doors
A list of 50 *Door* instances.

points
A list of 100 *Point* instances.

system
A single *System* instance.

timers
A list of 100 *Timer* instances. First timer `timers[0]` is always started with the game, so if it's necessary to set initial variables and game state, use this approach:

```
if timers[0].value == 1:
    # Initialize here
    pass
```

viewport
A single *Viewport* instance.

yozhiks
A list of 10 *Yozhik* instances. First yozhik `yozhiks[0]` is player himself.

Note: All classes below cannot be instantiated in scenario, and, in fact, they're not in the scope.

---

class Bot

ai
(*bool*) – should bot function on its own.

can_see_target
(*bool*, *read-only*).

goto
(*Point*) – make bot go to given *Point*.

level
(*int*) – a level of the bot, see *list of bot level constants* for possible values.

point
(*Point*, *read-only*) – a *Point* where bot is now.

target
(*Yozhik*) – attack target of the bot.

class Button

is_pressed
(*bool*, *read-only*).

press()

class Door

state
(*int*, *read-only*) – see *list of door state constants* for possible values.

open()

close()

class Point
Points are set in the map editor, and they are primarily used to tell a bot where to go. They can also be used to easily mark a location on map to serve as a trigger, or to display a message with *print_at()*.

pos_x
(*int*) – *x* coordinate of the point.

pos_y
(*int*) – *y* coordinate of the point.

class System

bots
(*int*) – number of bots.

color
(*int*) – color of *print_at()* messages.

It's a triple of 8-bit integers packed in one: `blue*65536 + green*256 + red`. It's easier to use *set_color()* instead of setting color value to this attribute.

Default color is `48128`, or `rgb(0, 188, 0)`.

frag_limit
(*int*) – see *list of frag limit constants* for possible values.

game_mode
(*int*, *read-only*) – current game mode, see *list of games modes* for possible values.

class Timer
A timer object that counts game ticks.

One game tick is roughly *1/50* of a second.

enabled
(*bool*) – is the timer ticking.

value
(*int*) – how much ticks did the timer count.

start()

stop()

class Viewport
Viewport object holds the location of top-left game screen corner in relation to top-left map corner.

pos_x
(*int*, *read-only*) – *x* coordinate of top-left screen corner.

pos_y
(*int*, *read-only*) – *y* coordinate of top-left screen corner.

class Yozhik

ammo
(*int*) – amount of ammo for current weapon.

armor
(*int*) – armor points.

frags
(*int*) – number of frags.

is_weapon_in_inventory
(*bool*) – setting *is_weapon_in_inventory* to `True` places current weapon in yozhik's inventory.

health
(*int*) – health points.

pos_x
(*float*) – *x* coordinate of yozhik's position.

pos_y
(*float*) – *y* coordinate of yozhik's position.

speed_x
(*float*) – *x* coordinate of yozhik's speed vector.

speed_y
(*float*) – *y* coordinate of yozhik's speed vector.

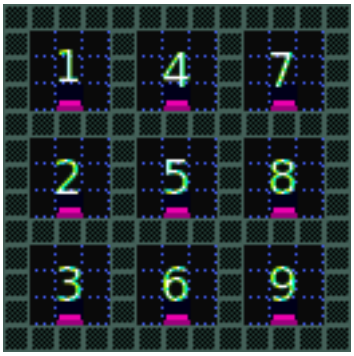
team
(*int*) – number of team.

view_angle
(*int*) – a value in range `[0, 127]`, when yozhik looks up it's 0, when he looks straight to the right or left it's 64, when he looks down it's 127.

weapon
(*int*) – current weapon, see *list of weapon constants*. Setting value to this attribute makes yozhik switch to the weapon, but does not place it in his inventory. If he didn't have it before and switches back, the weapon will be gone, unless `is_weapon_in_inventory` was set.

spawn (*point: int*)
Spawn yozhik in the given spawn-point.

Spawn points are enumerated starting at 1, from top to bottom, left to right:



Constants

Weapons:

W_BFG10K (*0*)

W_BLASTER (*1*)

W_SHOTGUN (*2*)

W_SUPER_SHOTGUN (*3*)

W_MACHINE_GUN (*4*)

W_CHAIN_GUN (*5*)

W_GRENADE_LAUNCHER (*6*)

W_ROCKET_LAUNCHER (*7*)

W_HYPERBLASTER (*8*)

W_RAILGUN (*9*)

Door states:

DS_CLOSED (*0*)

DS_OPEN (*1*)

DS_OPENING (*2*)

DS_CLOSING (*3*)

Frag limits:

FL_10 (*0*)

FL_20 (*1*)

FL_30 (*2*)

FL_50 (*3*)

FL_100 (*4*)

FL_200 (*5*)

Bot levels:

BL_VERY_EASY (*0*)

BL_EASY (*1*)

BL_NORMAL (*2*)

BL_HARD (*3*)

BL_IMPOSSIBLE (*4*)

Game modes:

GM_MULTI_LAN (*0*)

GM_MULTI_DUEL (*1*)

GM_HOT_SEAT (*2*)

GM_MENU (*3*)

GM_SINGLE (*4*)

GM_SHEEP (*5*)

GM_HOT_SEAT_SPLIT (*6*)

CHAPTER 4

Indices and tables

- genindex
- search

Index

R

S

T

V

W

Y