ThePythonGameBook Documentation

Release 0.4

Horst JENS

April 27, 2014

CONTENTS

Contents:

CHAPTER

ONE

GOBLIN VS. TESTDUMMY

1.1 about

This page shows how to formulate an idea (combat calculation) as a series of instructions and code it using python3.

1.2 idea

Imagine a young goblin named "Grunty". He dreams of being a fearsome warrior one day. However, at the moment, he is rather unskilled and barely capable of handling a weapon. Grunty goes to the training place fighting against a wooden testdummy.

Handling a weapon is no easy task, and Grunty will, despite his best efforts, not always land a blow. Sometimes he will miss.

Independent of the graphical representation (there is none yet), some combat mechanics must be formulated so that the computer can emulate a fight.

Let us assume that -like a `pen & paper` role player- Grunty has a low basic chance of hitting the testdummy while swinging his weapon. We express that as a low integer (whole) like 3. A more skilled warrior would have an higher attack value. A line of python code saying just that would look like:

```
grunty_base_attack = 3
```

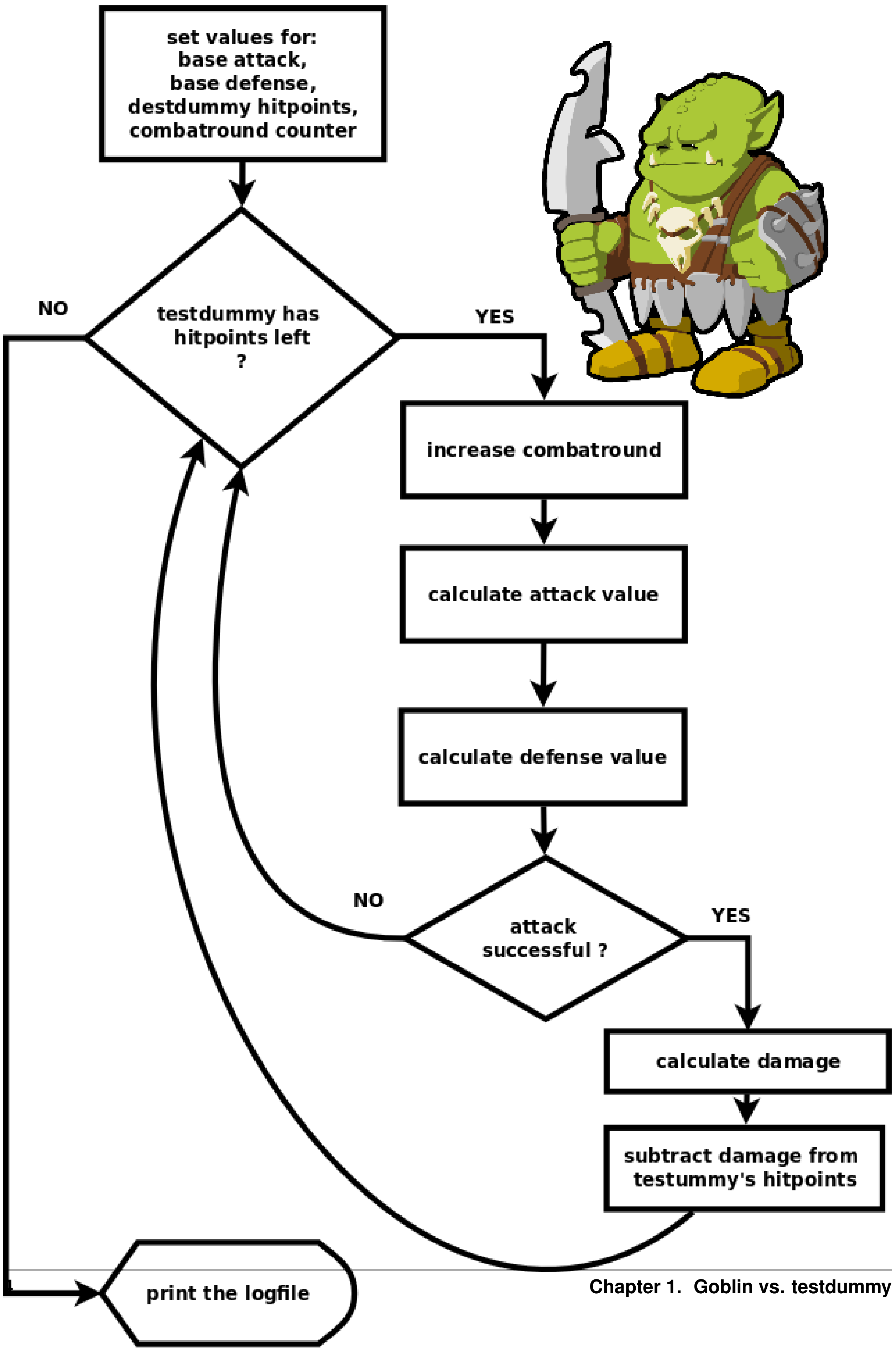
The part left of the equal sign is called a `variable` (because some time in the future hopefully Gruntys attack value will increase) and the part right from the equal sign is called a value. In this case, an `integer` value because we use whole numbers (no decimal point). Decimal numbers (like 0.3) are called `float` values. Note that in python, variable names can not include a space (python would think those are two different commands) and therefore an underscore _ is used. It is also possible to use *camelCase* like: `gruntyAttack` instead. By convention, all variable names begin with a lowercase character.

But not every strike is the same for Grunty. Sometimes he has luck and performs a perfect attack, sometimes not. To reflect this variety, let us add some random values on top his base chance, to reflect the actual luck / skill / performance / difficulty of combat. To calculate the outcome of one single attack in a combat situation, some random value is added to the base attack value. In pen & paper role playing games, the player has usually to throw two dice and add the number of eyes rolled to the base attack value.

```
attack_value = grunty_base_attack + random_integer_value
```

The resulting attack_value is now hopefully greater than the base attack value. Let it compare to a similar calculated defense value to see if Grunty scored a hit.

Grunty is fighting against a wooden, unmovable testdummy. The testdummy is very easy to hit: We give it a extreme low defense value of 1, reflecting instrinct defense abilitys like diffictult-to-hit geometry and shape.


set values for:
base attack,
base defense,
destdummy hitpoints,
combatround counter
testdummy has hitpoints left ?
NO
YES
increase combatround
calculate attack value
calculate defense value
attack successful ?
NO
YES
calculate damage
subtract damage from testummy's hitpoints
print the logfile

```
base_defense = 1
```

In actual combat, sometimes even an heap of wood can get lucky... not by actual moving, but maybe the sun is in his rear, blending the attacker, maybe the wind and the rain work in his favor. So to calcualte actual combat performance, let us add some random value on top of the (low) base defense value, but less than for the attacker. Let's say only rolling one die is allowed to roll and added to the base defense.

```
defense_value = base_defense + random_integer_value
```

The combat runs in several combat rounds: In each combat round, the computer calculates the attack and defense values (using his random generator functions) and compares those two. If the attacking value is greater than the defending value, Grunty managed a hit on the testdummy. If both values are equal, Grunty manages a "glancing blow", a hit that does no damage. If the attack value is greater than the defense value, damage is calculated as an random number between 0 and 10. With dice, this could be calculated by rolling 2 dice and subtracting 2 from the sum of eyes.

The damage is then substracted from the "Hitpoints" (also an integer number) of the testdummy, and the whole process is reapeated until the testdummy has no hitpoints left.

The fresh, undamaged testdummy must start with an given number of hitpoints, say 200 hiptoints. Also it is interesting to know how many combat rounds Grunty needs to destroy the testdummy, so a combat round counter is needed and set to 0 (you will soon see why) before the game.

```
testudmmy_hitpoints = 200
combat_round_counter = 0
```

If you have read the instructions above you have basically read a program (that is a set of instructions) and you could now simulate Gruntys combat performance by using those instructions, a pair of dice and paper and pen to keep track of combat rounds and hitpoints.

To let a computer do this task instead, a set of commands and concepts is necessary:

- *Variables*: numbers that can have different values over time, like the testdummys hitpoints or the round counter
- *Operators* to compare two variables
- *Control structures*:
  - a *Conditional* `if` statement to decide what to do if the attack value is greater than the defense value
  - a *Loop* to repeat the whole process as long as necessary
- Some random generator *functions*

CHAPTER

TWO

CODE

2.1 prerequesites

- necessary:
  - python3 installed
  - text editor can save python files (like *goblindice001.py*)
  - python3 interpreter can launch saved python file.
- recommended:
  - python-friendly IDE like IDLE, Geany etc.
  - ability to read and type (blind typing) using the 10 finger system, instead of copy and paste
  - python shell to lookup commands
  - executing python with filename as parameter from the command line *python3 goblindice001.py*)

2.2 source code

```
"""
Name:              goblindice001.py

Purpose:           teaching basic python: while, if, elif, else, +=
idea:              simple combat sim of one goblin against a test dummy
edit this code:    https://github.com/horstjens/ThePythonGameBook/
                   blob/master/python/goblindice/goblindice001.py
edit tutorial:     https://github.com/horstjens/ThePythonGameBook/
                   blob/master/goblindice001.rst
main project:      http://ThePythonGameBook.com
Author:            Horst JENS, horst.jens@spielend-programmieren.at
Licence:           gpl, see http://www.gnu.org/licenses/gpl.html
"""
import random

# Grunty, the untrained goblin, has some attack skill
grunty_attack = 3   # integer value

# the wooden, unmovable testdummy has poor defense, but many hitpoints
testdummy_hitpoints = 200 # integer value
testdummy_defense = 1

```

```
logfile = " Grunty vs. the wooden testdummy"    # string
combatround = 0

while testdummy_hitpoints > 0:
    combatround += 1         # the same as: combatround = combatround +1
    logfile += "\n*** Round: " + str(combatround) + " ***" # \n new line
    logfile += ", target has {} hitpoints".format(testdummy_hitpoints)
    attack= grunty_attack + random.randint(1,6)+random.randint(1,6)
    defense = testdummy_defense + random.randint(1,6) # roll one die
    if attack > defense: # did Grunty hit the testdummy ?
        logfile += "\nSmack! Grunty hits his target with a most "
        logfile += "skilled attack: {} > {}".format(attack, defense)
        damage = random.randint(1,6)+random.randint(1,6)-2 # 0-10 damage
        testdummy_hitpoints -= damage    # subtract damage from hitpoints
        logfile += "\n...and inflicts {} damage!".format(damage)
    elif attack == defense:
        logfile += "\nGrunty manages to nearly hit the target, but "
        logfile += "he makes no damage {0} = {0}".format(attack)
    else:
        logfile += "\n Oh no! Grunty does not even hit his target "
        logfile +=  "{} < {}".format(attack, defense)
logfile += "\n" + "- " * 20 # make a dashed line by multiplying a string
logfile += "\nVictory for Grunty after {} rounds".format(combatround)

print(logfile)
```

2.3 output

Some example output:

```
*** Round: 22 ***, target has 104 hitpoints
 Oh no! Grunty does not even hit his target 5 < 9
*** Round: 23 ***, target has 104 hitpoints
Grunty manages to nearly hit the target, but he makes no damage 9 = 9
*** Round: 24 ***, target has 104 hitpoints
Smack! Grunty hits his target with a most skilled attack: 6 > 4
...and inflicts 2 damage!
*** Round: 25 ***, target has 102 hitpoints
Oh no! Grunty does not even hit his target 6 < 9
```

2.4 code discussion

Some elements in this code may need explaining:

line number	term	explanation
1 - 13	*docstring*	Some multi-line text *string* insinde triple-quotes `"""`. If a docstring is the first statement in a file it is automatically stored in the gloabel variable `__doc__`. Docstrings are interpreted by python as something interested for humans only, like a *comment*. Docstrings are not necessary, but nice to have.
14	`import random`	To make use of any *functon* in pythons random module, it is necessary to import this module first. Later in this code functions of the random module will use the prefix `random.`.
16	`comment`	Everything behind a `#` sign in python is a comment. Comments are useful for human eyes only and always ignored by Python.
17	`assign, inline comment`	The value of 3 is assigned to the variable `grunty_attack`. (You best read it from right to left). Note that the part right from the `#` sign is also a comment.
26	`loop, expresssion`	The `while keyword` indicates, together with the colon at the end of this line, the beginning of an `indented code block`. This code block is reapeated as long as the `expression``right from ``while``remains ``True`.
27	`increment`	The value of the the `variable combatround` is incremented by 1. Because this is made even before some combat values are calculated, `combatround` was set to 0 before the while loop.
28	`strings and numbers`	Another textstring is appended to the textstring variable `logfile`. The starting new line sign (`\n`) is a so called `escaped character`. Because `combatround` is of type integer, it can *not* be appended to the textstring `logfile`. Therefore, ‘’combatround” must first be converted into a string using the `str()` function.
29	`format mini language`	Using Format String Syntax instead of the `str` function, the curly brackets are replaced by python with the expression inside the round brackets of `.format()`. See https://docs.python.org/2/library/string.html
30	`random.randint()`	The `random.randint()` function generates a (nearly) random integer between (including) the first (lower) and the second (higher) number in the round brackets. Writing instead `random.randint(1,6)*2` or `random.randint(2,12)` would be similar, but not the same. See next page for more on this topic.
32	`conditional, if`	Like the `while` keyword, the `if` keyword need an expression and a colon and is followed by an `idented code block`. This code block will only be executed if the `expression` is `True`. Note that you can write an inline-comment after the colon with the `#` sign. For python, the comment is ignored and the line ends with the colon.
33 - 34	`strings`	Note that the string in line 33 ends with an space and the string in line 34 does not begin with a new line. In fact, line 33 and line 34 can be written in lone (very long) line. In this case, two seperate lines were written for layout reasons, to make no code line longer than 72 characters. See http://legacy.python.org/dev/peps/pep-0008/
34	`format mini language`	The first pair of curly brackets get replaced by the first value inside `.format()`, the second pair of curly brackets get replaced by the second value inside `.format()` and so on.
35	`random.randint()`	Here,‘‘random.randint(2,12) could have be used instead, and would have resulted similar, bot not exact the same distribution of random values as this formular. See next page for more details.
36	`decrement`	Like the increment, this line could have been written as: `testdummy_hitpoints -= testdummy_hitpoints - damage`. But why type more than necessary?

pause

line number	term	explanation
38	`elif`	Each `if` code block can have many (including none) `elif` code blocks. Like the `if` keyword, the `indented code block` after elif will only be executed if the `expression` was False for all previous `elif` expressions as well as for the initial `if` and only if the actual expression (right of `elif`) is `True`.
38	`equal test`	Please note that the == operator is used to *test for equality*. The = operator assigns values.
40	`format mini language`	A special trick using `.format()`: When you have one variable at several places inside a string, like in this case, you can use their number inside the curly bracket. Python always start counting with 0, so the first variable in the round brackets is referred as `{0}`, the second variable as `{1}` and so on.
41	`else`	If the `if` and all `elif` expressions are `False`, the `idented code block` behind the `else:` keyword is executed. The `else:` keyword itself is optional and need no expression behind it.
44	`indentation`	This line is *outside* the `while block` ! Take a close look at the previous line with the same indentation: It's line 26. That means that this line will only executed if the expression in line 26 becomes `False`
46	`empty line`	There is no specific reason to let this line empty: it's just to have the code pretty layouted. Python ignores empty lines.
47	`print()`	The `print()` keyword can output any variable to the screen. Note that since python version 3, `print` need round brackets.

next page: functions !

CHAPTER

THREE

GOBLIN DICE TUTORIAL

This is the goblin dice tutorial starting page.

This document is meant to give a tutorial-like overview of all common tasks while using Sphinx.

The green arrows designate “more info” links leading to advanced sections about the described task.

let us start :

3.1 boring welcome headline

with some not necessary text below it

3.2 second boring sub-headline

also with non-relevant text

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*