SHONNER PRESS

pydice Operations Manual

Release 3.12.0 (Third Printing)

Shawn Driscoll

12/12/2022

CONTENTS

CHAPTER

ONE

OVERVIEW

This documentation explains how to install and use the **pydice module** for your gaming projects.

pydice is easy-to-use open source die rolling software. Written in Python 3.11.0 and using a variety of IDEs, **pydice** supports many gaming and RPG die rolling conventions.

pydice also supports logging, error reporting, and debugging of rolls made.

The free-to-use source is available at its GitHub repository.

CHAPTER

TWO

INTRODUCTION

2.1 Preface

Back during the release of **diceroll 2.2**, I wanted to learn something new in regards to Python. Even though I was using 2.5.4, there was still a lot about it that I have never delved into. Sphinx was something I had not really paid any mind to in the past. It was yet another one of those *need to know only* things about Python. Some things I'd get around to learning only when I had to, but only if it was part of something else that I had taken an interest in doing.

So somewhere in my discovering of PyMongo, I had been pointed to Sphinx and Jinja. They were both something about document generation. And since I had just learned about Pandas and CSV, I was in a data retrieval mood still.

In a nutshell, Sphinx is an EXE (generated during its install from a pip command, which is still magic to me how *it just runs* in Python 3.9+) that generates documents. Nothing too fancy. Just simple documents that could be read easily/quickly through any device using any viewer. And when I learned that Sphinx could read Python modules and produce documents from their `.__doc__` strings, I knew I just had to spend a couple days learning how all that stuff happens.

So basically, my Python dice rolling module has its own operations manual now. And some rabbit holes are worth their going into.

-Shawn

2.2 Requirements

- **Microsoft Windows**

  pydice has been tested on Windows versions: 10. It has not been tested on MacOS or Linux.

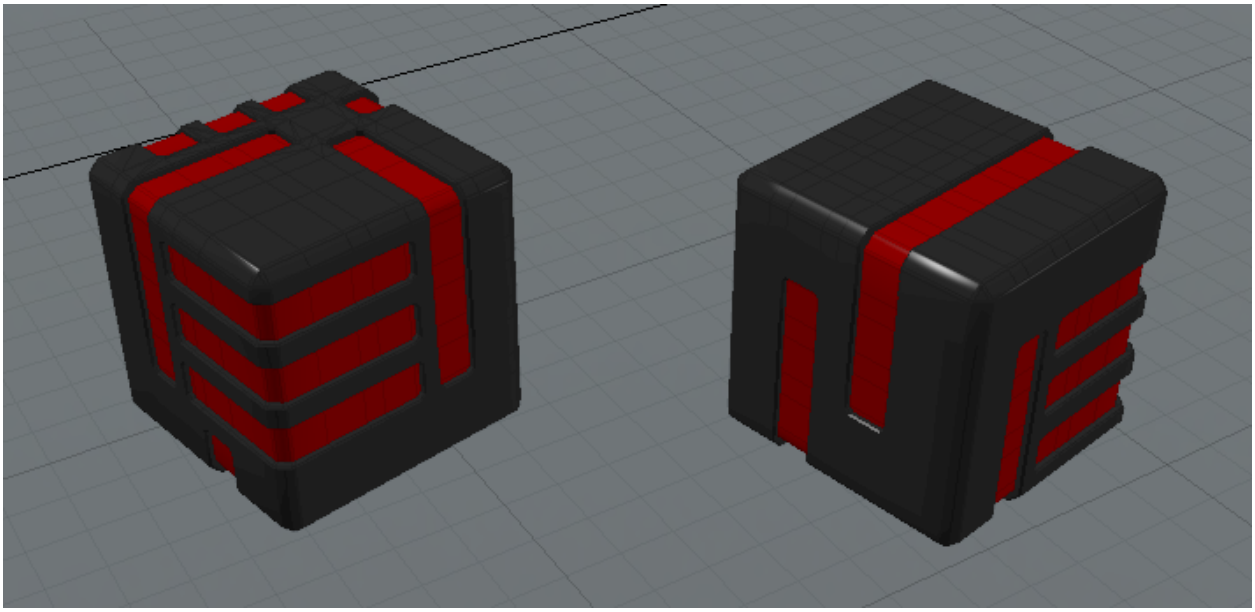
- **Python 3.11**

  pydice was written using the C implementation of Python version 3.11. Also known as CPython. With some doing, this module could of course be re-written for Jython, PyPy, or IronPython.

  Eclipse/PyDev, PyCharm, NetBeans, and IDLE all work fine for running this module. One of the easiest to use is Microsoft Visual Studio Code.

- **Your Game**

  pydice can be used as a standalone program. But where it shines though is when it's imported into a game of yours.



2.3 Installing Locally to Your Folder

Installing **pydice** is as easy as always. Just copy `pydice.py` into the same folder your code happens to be in.

Then add this line at (or near) the top of your code:

```
from pydice import roll
```

2.4 Installing as a Package

If your code setup is different, in that you like to keep your function modules in a folder separate from your main code, you could copy `pydice.py` into that folder.

Say you have a folder called `game_utils`, and assuming you have an `__init__.py` inside it, just copy `pydice.py` into your `game_utils` folder and add this line near the top of your code:

```
from game_utils.pydice import roll
```

Some ways to see if the `pydice` module was installed correctly is by typing:

```
>>> print(roll('info'))
('3.12', 'roll(), release version 3.12.0 for Python 3.11.0')
>>> print(roll.__doc__)
    The dice types to roll are:
        '4dF', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D8', 'D09', 'D10', 'D12', 'D20',
        'D30', 'D099', 'D100', 'D0999', 'D1000', 'D44', 'D66', 'D666', 'D88', 'DD',
        'FLUX', 'GOODFLUX', 'BADFLUX', 'BOON', 'BANE', 'ADVANTAGE',
        'DISADVANTAGE', and also Traveller5's 1D thru 10D rolls
    Some examples are:
    roll('D6') or roll('1D6') -- roll one 6-sided die
    roll('2D6') -- roll two 6-sided dice
    roll('D09') -- roll a 10-sided die (0 - 9)
    roll('D10') -- roll a 10-sided die (1 - 10)
    roll('D099') -- roll a 100-sided die (0 - 99)
    roll('D100') -- roll a 100-sided die (1 - 100)
    roll('D66') -- roll for a D66 chart
    roll('FLUX') -- a FLUX roll (-5 to 5)
    roll('3D6+6') -- add +6 DM to roll
    roll('4D4-4') -- add -4 DM to roll
    roll('2DD+3') -- roll (2D6+3) x 10
    roll('BOON') -- roll 3D6 and keep the higher two dice
    roll('4dF') -- make a FATE roll (-4 to 4)
    roll('4D') -- make a Traveller5 4D roll
    roll('4D6H3') -- roll 4D6 and keep the higher three dice
    roll('3D6L2') -- roll 3D6 and keep the lower two dice
    roll('info') -- release version of program
    roll('2D8 # weapon damage') -- a 2D8 roll with a comment added
    An invalid roll will return a -9999 value.
```

CHAPTER

THREE

WHAT'S NEW WITH PYDICE?

3.1 New in pydice 3.12.0

pydice has been updated for **Python 3.11.0**.

New in pydice 3.11.8

FLUX, GOODFLUX, and **BADFLUX** will ignore **dice roll modifiers**.

Note: Dice roll modifiers are no longer ignored when using **H** or **L**.

New in pydice 3.11.7

pydice now checks for version of Python installed. Will display a warning on screen and in the log.

New in pydice 3.11.6

Error-trapping for invalid dice modifiers now.

New in pydice 3.11.5

The new **D1** roll generates a range of **0 - 1**.

D2 rolls now generate a range of **1 - 2**.

New in pydice 3.11.0

H and **L** have been added for keeping higher or lower dice.

`roll('3D6H2')` – roll 3D6 and keep the higher 2 dice.

`roll('2D20L1')` – roll 2D20 and keep the lower die.

Note: Dice roll modifiers are ignored when using **H** or **L**.

New in pydice 3.10.6

Comments can be entered with die rolls, such as `roll('2D8 # weapon damage')` Any comments used will appear in the `pydice.log` file.

New in pydice 3.10.5

The **D666** roll has been added. The result is D6*100 + D6*10 + D6.

New in pydice 3.10.0

The **D0999** roll has been added. It generates a range of **0 - 999**.

The **D1000** roll has been added. It generates a range of **1 - 1000**.

New in pydice 3.9.0

pydice now uses `int(random() * n + 1)` instead of `randint(1, n)` to generation its random numbers much faster.

New in pydice 3.8.0

PyDiceroll is now **pydice**. PyDiceroll has been discontinued. Modified **DEBUG** level logging for **BOON**, **BANE**, **ADVANTAGE**, and **DISADVANTAGE** rolls. The newly introduced default roll performs a **2D6** roll. Added error-trapping when performing **MINMAXAVG** rolls at the CMD prompt.

New in PyDiceroll 3.7.2

MINMAXAVG calculates negative averages correctly now.

New in PyDiceroll 3.7.1

A new secret roll has been added. This is a beta test of the Fate roll type where dice mods can be added. As well as number of Fate dice to roll.

New in PyDiceroll 3.7.0

ADVANTAGE and **DISADVANTAGE** rolls, for d20 systems, are now do-able.

New in PyDiceroll 3.6.0

PyDiceroll no longer requires **colorama**.

New in PyDiceroll 3.5.0

More than one **D09** can be rolled at a time now. Added to the **MINMAXAVG** list.

New in PyDiceroll 3.4.0

The **MINMAXAVG** roll has been added. Just doing:

```
>>> roll('MINMAXAVG')
```

will output the Min, Max, and Averages for various rolls. Mostly for testing. Nothing is returned from this roll. So print or variable assignment is not needed.

```
Test Roll: 1d2, Min: 0, Max: 1, Avg: 0.5, Sample: [0, 0, 0, 0, 1, 0, 1, 0, 1, 1]
Test Roll: 1d3, Min: 1, Max: 3, Avg: 2.0, Sample: [3, 3, 3, 3, 1, 3, 3, 3, 2, 1]
Test Roll: 1d4, Min: 1, Max: 4, Avg: 2.5, Sample: [3, 2, 3, 1, 1, 1, 1, 3, 1, 3]
Test Roll: 1d5, Min: 1, Max: 5, Avg: 3.0, Sample: [4, 1, 4, 3, 3, 1, 5, 3, 5, 1]
Test Roll: 1d6, Min: 1, Max: 6, Avg: 3.4, Sample: [2, 3, 4, 4, 1, 6, 3, 6, 2, 5]
Test Roll: 1d8, Min: 1, Max: 8, Avg: 4.5, Sample: [2, 6, 2, 4, 7, 8, 3, 8, 8, 6]
Test Roll: 1d09, Min: 0, Max: 9, Avg: 4.4, Sample: [9, 8, 1, 4, 3, 8, 6, 2, 6, 1]
Test Roll: 1d10, Min: 1, Max: 10, Avg: 5.6, Sample: [6, 4, 3, 9, 2, 8, 6, 9, 2, 2]
Test Roll: 1d12, Min: 1, Max: 12, Avg: 6.5, Sample: [11, 3, 5, 5, 2, 2, 6, 10, 4, 6]
Test Roll: 1d20, Min: 1, Max: 20, Avg: 10.5, Sample: [1, 17, 20, 18, 10, 14, 11, 2, 14, 7]
Test Roll: 1d30, Min: 1, Max: 30, Avg: 15.9, Sample: [28, 6, 30, 23, 21, 16, 5, 27, 11, 4]
Test Roll: 1d099, Min: 0, Max: 99, Avg: 50.3, Sample: [80, 50, 22, 70, 62, 15, 58, 48, 18, 32]
Test Roll: 1d100, Min: 1, Max: 100, Avg: 49.7, Sample: [74, 11, 95, 69, 99, 50, 82, 74, 78, 95]
Test Roll: 4df, Min: -4, Max: 4, Avg: -0.0, Sample: [-1, 3, 0, 0, -1, -1, -1, 2, -2, 2]
Test Roll: flux, Min: -5, Max: 5, Avg: -0.0, Sample: [-1, -3, -2, 3, 3, -1, 2, 0, 0, -1]
Test Roll: goodflux, Min: 0, Max: 5, Avg: 2.0, Sample: [3, 3, 5, 0, 1, 3, 1, 3, 0, 1]
Test Roll: badflux, Min: -5, Max: 0, Avg: -1.9, Sample: [0, -1, -2, -1, -2, -2, -5, 0, -2, -5]
Test Roll: boon, Min: 2, Max: 12, Avg: 8.4, Sample: [11, 6, 9, 7, 7, 12, 8, 10, 9, 9]
Test Roll: bane, Min: 2, Max: 12, Avg: 5.5, Sample: [6, 6, 5, 2, 2, 4, 5, 4, 8, 8]
Test Roll: 2d4, Min: 2, Max: 8, Avg: 5.0, Sample: [5, 6, 7, 8, 3, 3, 4, 5, 7, 8]
Test Roll: 3d4, Min: 3, Max: 12, Avg: 7.6, Sample: [7, 9, 6, 8, 10, 11, 6, 4, 7, 10]
Test Roll: 4d4, Min: 4, Max: 16, Avg: 10.0, Sample: [10, 8, 15, 12, 13, 7, 12, 10, 9, 13]
```

New in PyDiceroll 3.3.1

Fixed error if non-numbers are entered.

New in PyDiceroll 3.3.0

Input errors for `roll()` will now return a value of -9999 instead of 0.

New in PyDiceroll 3.2.1

New **D44** and **D88** rolls have been added. These are table rolls, similar to the **D66** roll.

Parsing

The `roll()` function has improved parsing that allows for spaces from other program sources. Error-checking understands this and will even check for negative numbers of dice. This improved feature works whether **PyDiceroll** is being used in a Python program or at a CMD prompt.

Refactored for Python 3.9

PyDiceroll's code has been updated from 2.5 to 3.9 standards.

The **D5** has been added to **PyDiceroll**. It is basically a **D10** divided by 2, much like how the **D3** die is a **D6** that is divided by 2.

CHAPTER

FOUR

PYDICE TUTORIAL

4.1 Rolling the Dice

Once `pydice.py` is installed and your code is able to import the module, its `roll()` function can be used right away. This function returns an integer, by the way. So it can be used as any other integer would be used. But first, we must give this function a value to work from.

roll(*dice*)

dice = a string of three ordered concatenated values:

number_of_dice + *dice_type* + *dice_roll_modifier*

As examples:
dice = '2' + 'D10' + '-2'
dice = str(3) + 'D6' + '+2'
dice = 'FLUX'

dice_roll_modifier must include a '+' or '-' with its value.

Note that both *number_of_dice* and *dice_roll_modifier* are optional, and may not even be used by some *dice_type* rolls. Leaving the entire string value empty or blank will default to making a '2D6' roll.

Note: **FLUX, GOODFLUX**, and **BADFLUX** will ignore **dice roll modifiers**.

Those of you that have used dice rolling programs before will notice that something is different. And that is, `roll()` uses a string for its input:

```
>>> die1 = roll('1D6')
>>> die2 = roll('1d6')
>>> dice = '3D4+1'
>>> print(die1, die2+4, roll(dice))
3, 6, 9
```

The return values from `roll()` are always integer.

Notice that the inputted string values can be upper or lower case.

The dice types to roll are:

> **D1, D2, D3, D4, D5, D6, D8, D09, D10, D12, D20, D30, D99, D100, D0999, D1000, D44, D66, D666, D88, DD, BOON, BANE, FLUX, GOODFLUX, BADFLUX, ADVANTAGE, DISADVANTAGE**, and **4dF**.

D09 rolls will generate a range of **0 - 9**.

D99 rolls will generate a range of **0 - 99**.

D1 rolls will generate a range of **0 - 1**.

The **4dF** roll type is for FATE mechanics. A range of **-4 to 4** is generated.

Traveller5 uses **1D** thru **10D** rolls, depending on the difficulty of a task. DMs are supported.

ADVANTAGE and **DISADVANTAGE** are dice rolls for the d20 system.

D0999 rolls will generate a range of **0 - 999**.

D1000 rolls will generate a range of **1 - 1000**.

D666 rolls result in D6*100 + D6*10 + D6.

Comments can be added to dice type strings, such as `roll('2d10 # Strength Attribute)'` The comments will appear in the log. This is to help document what certain rolls were used for.

Note: You may recognize some of these dice types from various tabletop role-playing games. Not all dice types are covered by **pydice**. However, more are planned for in future releases.

---

pydice uses a simple standard when it comes to rolling various dice types.

Some examples are:

```
roll('D6') or roll('1D6') # roll one 6-sided die
roll('2D6') # roll two 6-sided dice
roll('D09') # roll a 10-sided die (0 - 9)
roll('D10') # roll a 10-sided die (1 - 10)
roll('D099') # roll a 100-sided die (0 - 99)
roll('D100') # roll a 100-sided die (1 - 100)
roll('D66') # roll for a D66 chart
roll('FLUX') # a FLUX roll (-5 to 5)
roll('3D6+6') # add +6 DM to roll
roll('4D4-4') # add -4 DM to roll
roll('2DD+3') # roll (2D6+3) x 10
roll('BOON') # roll 3D6 and keep the higher two dice
```

(continues on next page)

(continued from previous page)

```
roll('4dF') # make a FATE roll (-4 to 4)
roll('4D') # make a Traveller5 4D roll
roll('4D6H3') # roll 4D6 and keep the higher three dice
roll('3D6L2') # roll 3D6 and keep the lower two dice
roll('2D8 # weapon damage') # a 2D8 roll with a comment added
```

pydice can be used directly at a CMD prompt using:

The long form:

```
C:\>pydice.py roll('2d6-2')

Your '2D6-2' roll is 10.
```

Or the short form:

```
C:\>pydice.py 2d6-2

Your '2D6-2' roll is 7.
```

Note: Typing `pydice.py -h` will provide some help.

A **TEST** roll that calculates percentages for 2D6 can be issued:

```
>>> roll('test')
      6x6 Roll Chart Test
     1    2    3    4    5    6
1  262  296  250  292  292  241
2  270  315  299  236  279  261
3  295  274  288  274  291  295
4  273  284  279  276  249  273
5  293  280  291  276  280  283
6  270  276  282  272  273  280
            6x6 Roll Chart Percentage
        1       2       3       4       5       6
1   2.62%   5.66%   8.60%  11.38%  13.93%  16.23%
2   5.66%   8.60%  11.38%  13.93%  16.23%  13.95%
3   8.60%  11.38%  13.93%  16.23%  13.95%  11.02%
4  11.38%  13.93%  16.23%  13.95%  11.02%   8.25%
5  13.93%  16.23%  13.95%  11.02%   8.25%   5.56%
6  16.23%  13.95%  11.02%   8.25%   5.56%   2.80%
```

The roll will return a list of percentages for 2-12 rolled.

CHAPTER

FIVE

USING ROLL() IN YOUR OWN CODE

5.1 For Simple Die Rolls

Sample Outputting of Die Rolls:

```
# import the roll() module
from pydice import roll

# enter the roll type to be made
number_of_dice = input('Number of dice to roll? ')
dice_type = input('Dice type? ')
dice_roll_modifier = input('DM? ')

# make sure that there is a plus or minus sign in the DM string
if dice_roll_modifier[0] != '-' and dice_roll_modifier[0] != '+':
    dice_roll_modifier = '+' + dice_roll_modifier

# concatenate the values for the dice string
dice = number_of_dice + dice_type + dice_roll_modifier

print()
print('Rolling', dice)

# do 20 rolls
for i in range(20):
    print('You rolled a %d' % roll(dice))
```

5.2 For Probabilites

Sample Task Resolution:

```
# import the roll() module
from pydice import roll

# Enter your character's chances to succeed at a task
skilled = input('Is your character trained for the task ([y]/n)? ')
if skilled == 'n':
    die_mod = -3
else:
    print("Enter your character's skill level")
    die_mod = int(input('(0 to 4)? '))
print('Enter the difficulty of the task')
difficulty = int(input('(Simple: +2 to Impossible: +16)? '))

# The player must roll the difficulty or higher for their character to succeed
dice_roll = roll('2D6') + die_mod
print()
print('You rolled:', dice_roll)
if dice_roll >= difficulty:
    print('Your character succeeds with the task.')
    if dice_roll - difficulty >= 6:
        print('Your character saved everyone.')
else:
    print('Your character fails at the task.')
    if dice_roll - difficulty < -3:
        print('Your character becomes injured.')
    if dice_roll - difficulty < -6:
        print('Your character died from injuries!')
```

5.3 For Repairing Game Code

Often times, game code will be downloaded or found that contains incorrect `randint()` calls for rolling two 6-sided dice. A line such as:

```
world_size = randint(2, 12) - 2
```

Easily becomes:

```
world_size = roll('2d6-2')
```

5.4 Encountering Errors

Entering an invalid string for `roll()` will return an error message, as well as a value of -9999 from the function:

```
print(roll('1d7'))
```

Error: ** DICE ERROR! '1D7' is unknown **

-9999

CHAPTER

SIX

DEBUGGING PYDICE

pydice has the ability to log rolls made during its execution. You will find `pydice.log` in the `Logs` folder it creates if one isn't there already. By default, this feature is kept to a minimum to allow **pydice** to execute at its fastest. **pydice** uses a default logging mode of `WARNING` which isn't that verbose.

```
dice_log.setLevel(logging.WARNING)
```

Setting the logging mode to `INFO` will record more info in the log.

```
dice_log.setLevel(logging.INFO)
```

Your **INFO** logging will output as:

…INFO pydice - Logging started.
…INFO pydice - roll() v3.12 started, and running…
…INFO pydice - '3D4' = 3D4+0 = 10

Changing **pydice's** logging mode to `DEBUG` will record debugging messages in the `Logs\pydice.log` file.

```
dice_log.setLevel(logging.DEBUG)
```

Your **DEBUG** logging will output as:

…INFO pydice - Logging started.
…INFO pydice - roll() v3.12 started, and running…
…DEBUG pydice - Asked to roll '3D4':
…DEBUG pydice - Using three 4-sided dice…
…DEBUG pydice - Rolled a 4
…DEBUG pydice - Rolled a 2
…DEBUG pydice - Rolled a 2
…INFO pydice - '3D4' = 3D4+0 = 8

Warning: Running **pydice** in `DEBUG` mode may create a log file that will be too huge to open. A program of yours left running for a long period of time could create millions of lines of recorded log entries. Fortunately, `pydice.log` is reset each time your program is run.

Note: Any errors encountered will be recorded as `ERROR` in the log file, no matter which logging mode you've chosen to use.

If your own code has logging enabled for it, be sure to let **pydice** know by changing `your_logger_function_here` to the name of the logger function used by your program that is calling `roll()`. The original line in **pydice** looks like this:

```
log = logging.getLogger('your_logger_function_here.pydice')
```

So, if your own code has:

```
log = logging.getLogger('dungeoneer')
```

then in **pydice**, make

```
log = logging.getLogger('dungeoneer.pydice')
```

CHAPTER

SEVEN

ALTERNATE PYDICE DISTRIBUTIONS

Often times, the **pydice** module is found in other formats. You may already have a copy of **pydice** that was distributed with another program you're using.

Besides its common `.py` format, **pydice** can be found in a `.pyd` format as well. This format is packaged as a dynamic link library, and will work the same way as the `.py` format. The format is typically bundled with the software that it was designed for.

Note: The `.pyd` format can only be imported and will not execute at a CMD prompt.

CHAPTER

EIGHT

SOFTWARE TITLES THAT USE PYDICE

Here is a sample list of software titles, at the time of this writing, using **pydice**:

CORE Micro DieRoller

Graphical Dice Roll

PyImperial CharGen

pyqtgraph_PyTravCalc

Python Feral Ghoul FSM

PyTravCalc

PyTravLITE

SectorGen

TPS DieRoller

TravEncounters

CHAPTER

NINE

DESIGNER'S NOTES FROM PYDICEROLL

9.1 In the Beginning

One of the first things I do when learning a new language is to discovery how it generates random numbers. Older computer languages from the '70s had their own built-in random number generators. Technically, they were pseudo-random number generators. But technically, I wanted to program my Star Trek games anyway no matter what they were called.

In the '80s, I would discover that not all computer languages came with random number generators built in. Many didn't have such a thing unless some external software library was installed. Both FORTRAN and C couldn't do random anything out of the box. A math library had to be picked from the many that were out there. And if none were available, a computer class on campus was available to teach you how to program your own random number generator from scratch.

By the '90s, random number generators were pretty much standardized as for as how accurately random they were. And they were included in standard libraries for various languages. By the time Python was being developed, the C language used to write Python had very robust random number generators. And because Python was written in C, it just made sense for it to make use of C libraries.

For those that are curious, **PyDiceroll** uses the `random.randint()` module that comes with CPython. There are stronger random generators out there now, with NumPy being one of them. But at the time of designing **PyDiceroll**, I didn't quite understand how-all NumPy worked, or what version of it to install. And for rolling dice, the built-in random number generator would be just fine.

9.2 Lessons Learned

In the past, when I needed a random number from 1 to say 6 (see 6-sided dice), I would use `INT(RND(1)*6) + 1`. And I would be used to doing it that way for probably 15 years or so, because that is how most BASIC languages did things. Other languages like C required me to whip out the 80286 System Developer's 3-ring binder to find out how `srand()` and `rand()` worked, and under what circumstances.

Fast-forward 20 years, and I'm learning CPython without knowing the difference between a CPython or an RPython or any other Python out there. I figured Python was the same all over, even though I had a feeling Linux did things differently because of its filepath naming and OS commands. And of course, the first thing I had to try was Python's `random` module, as well as its ugly-looking `randint()`.

Right away I noticed the way Python "loaded" modules was going to be a learning experience. I hadn't really programmed anything huge since my TANDY Color Computer 3 days running OS-9 Level II and programming in BASIC09 (https://en.wikipedia.org/wiki/BASIC09). Python would reveal different ways of importing modules the more I read about them, and the more code I poured over.

I would soon find that:

```
import random

print(random.randint(1, 6)) # roll a 6-sided die
```

Was the same thing as:

```
from random import randint

print(randint(1, 6)) # roll a 6-sided die
```

Which looked a bit cleaner. But I was debating if I wanted to use `randint()` at all in my normal coding.

So while I was learning how to write my own functions, as well as how to go about importing them, I came up with an idea for **PyDiceroll**. It would include a `roll()` function, and a `die_rolls()` function as a "side effect." Even though `die_rolls()` had no error-checking, `roll()` would call it after doing its own error-checking.

I was trying to avoid using:

```
from PyDiceroll import die_rolls

print(die_rolls(6, 2)) # roll two 6-sided dice
```

For my dice rolls, I wanted something more readable. Something like:

```
from PyDiceroll import roll

print(roll('2D6')) # roll two 6-sided dice
```

It was almost less typing, which I thought was great because I was going to be typing this function a lot for a Python project I had in mind. And it would be a lot easier to spot what kind of rolls were being made in my code. And the simple addition or subtraction of DMs to such a roll was making the function more appealing:

```
print(roll('2D6+3')) # roll two 6-sided dice and add a DM of +3 to it
```

9.3 The Channel 1

diceroll was written years ago. The Classic Python 2.5 code was used by both my TravCalc and TravGen apps, and got looked at by GitHub visitors who would google-by now and again. But not many programmers will ever use the code because of the simple fact that Python is now version 3.9+ something. So **diceroll**, along with a slew of other pre-Python 2.6 era modules, are the Channel 1 stations in the room that no TV can possibly watch.

It really comes down to a philosophy. I waited on learning Python until a version was released where I could say, *"This is Python."* Or say, *"This is what Python should be."* Something like that. Well… Python 2.5.4 was my Python.

I once said, *"I believe the next great computer programming language will be the one that remains true to its nature/design as it grows. And doesn't split the party as it grows."* I hung onto Python 2.5.4 for as long as possible. For a good fifteen years. Or I should say for a *great* fifteen years. Because they were great. But most great things come with an ending to them.

And so it was, that yesterday I would uninstall Python 2.5.4 along with all its things. And today, I would begin the installation of Python 3.9.4. I guess one could say it was the terminated support for Python 2.x this year that nudged me, along with some of the older Python 3.x terminations as well. Even Python 3.9+ saw earlier Python 3's as dead weight (Python 3's that I didn't want to deal with either), such as 3.0, 3.1, 3.2, 3.3, 3.4, and 3.5. And now they are gone. And I can skip ahead to a refined version of Python 3 with no baggage from 2.6 or 2.7 whatsoever.

Shawn Driscoll
April 23rd, 2021
US, California

CHAPTER

TEN

DESIGNER'S NOTES FROM PYDICE

10.1 Fast-Forward Some Years

Well, **pydice** (in all its various forms) has been out since 2014. It seems like only a few weeks ago still. A lot has happened in eight years. During that time, I have programmed pretty much in just Python. I have moved from one IDE to another. The current one I'm using is Microsoft Visual Studio Code. I'm still running on Windows 10 because my CPU is not compatible with Windows 11. And I have Python 3.9.13 installed (I have not jumped to version 3.10 or 3.11 because my *site-packages* is mostly targeted for 3.9. I don't want to risk updating to 3.11 and find out that my `pip list` is all for naught.

Since I no longer have Python 2.5.4 (a lost love of mine) installed, I can only run code written for it if I had made .EXE files of it. Py2exe was shear greatness for doing that at the time. I still make .EXE for my Python 3.9 code. But with PyInstaller. It's obviously a far-improved marvel to watch execute.

At the time of this writing, Python is the number one programming language being both used and learned. Only a year or two ago, I thought Python would still be relegated to the number three spot. But no. The more new programming languages that are released, the more popular Python becomes. Python is everywhere (except for in some phones in people's hands). But all-in-all, Python is everywhere. I don't think Python comes pre-installed yet on Windows systems. But a lot of Windows programs come with Python terminals built into them. I still have Poser 6 and Vue 9.5 Infinite running on my computer. And they both have some version of Python running under their hoods.

Very recently, I looked into Lua and Ruby. I installed both of them and gave them a try. It's really interesting to me how old these programming languages are. Python is over 30 years old. When I look at the TIOBE Index (https://www.tiobe.com/tiobe-index/), it's really an eye-opener as to what's popular in the computer programming field. A lot of the languages I used decades ago are now in the *under 1%* use. As I've said before on several occasions, *"I wish Python was around in the '70s!"*

10.2 Two Camps

So what has happened with dice programs in general for Python?

I've notice two worlds when it comes to programming.

By the way, programming is now called coding for some reason. It's similar to how creative writing is called typing.

Terms are dumbed down in order to not scare away potential students. Schools want to make money after all. Today's schools are certainly doing a good job of copy/pasting their lessons. YouTube has endless video selections on how to code dice rolling programs. They all use a variation of:

```
diceroll = random.randint(1, 6)
```

or:

```
diceroll = random.randrange(1, 7)
```

or:

```
diceroll = int(random.random() * 6 + 1)
```

for rolling 6-sided dice in their code, which is done procedurally for the most part.

Over on GitHub however, a lot of the dice apps are coded in functional and/or object-oriented fashion.

Nothing seems to be called a program anymore. Nearly everything is referred to as an app now.

These apps often code their dice rolls as:

```
def diceroll(k, n, dice_sides=6):
   counter = {n : 0 for n in range(k, k*dice_sides + 1)}
```

or as:

```
class Die(object):
   def __init__(self, sides = 6):
     self.sides = sides

   def roll(self):
     return randint(1, self.sides)
```

Neither of these is very appealing to me. The functional style is hard to read, while the object-oriented style creates objects when all is needed are random numbers.

GitHub has other styles of code, such as:

```
diceroll(1,6,2,3)
```

Which typically means something along the lines of roll 2 dice numbered 1-6 and add a die-modifier of 3 to the result. Again, it was hard to read some of the apps coded this way. I blame GitHub for propagating such code via their forking. Everyone is copy/pasting over there.

10.3 I Still Propose

It might just be me (in fact, I am sure of it), but I still prefer to use:

```
roll('2d6+3')
```

and:

```
roll('d20')
```

in my games. That will never change.

Shawn Driscoll
July 19th, 2022
US, California

CHAPTER

ELEVEN

PYDICE MODULE

pydice.**roll**(*str(number_of_dice) + str(dice_type) + str(dice_roll_modifier)*)

`roll()` accepts a **string value** made up of three concatenated values, then returns an integer.

String value comes from *number_of_dice* + *dice_type* + *dice_roll_modifier*

Some examples are:
'2' + 'D10' + '-2'
str(3) + 'D6' + '+2'
'FLUX'

dice_roll_modifier must include a '+' or '-' with its value.

Note that both *number_of_dice* and *dice_roll_modifier* are optional, and may not even be used by some *dice_type* rolls. Leaving the entire string value empty or blank will default to making a '2D6' roll.

Note: FLUX, GOODFLUX, and **BADFLUX** will ignore **dice roll modifiers**.

CHAPTER

TWELVE

OPEN SOURCE

12.1 MIT License

LICENSE AGREEMENT

12.2 Contact

Questions? Please contact shawndriscoll@hotmail.com

CHAPTER

THIRTEEN

FFE AGREEMENT

CHAPTER

FOURTEEN

ABOUT THE AUTHOR

Shawn Driscoll is an American artist. Computers are his main creation tool. His many hobbies are in sync with his being a student of all sciences. Some of which are discussed in length on his YouTube channel.

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- search

INDEX