Magic Squares (Python)
Knuth, *TACOP*, vol.1 3.3.2 ex.21

Magic Squares

Back in the days before MMORPGs, facebook and fizzy-drinks[1], children would spend many happy hours constructing magic squares (Well, maybe it was just me ...). Magic squares are, disappointingly, not magic at all. But they do have the exciting property that all their rows and columns, plus the two main diagonals, add up to the same number.

We are going to deal here with magic squares of order n, where n is an odd number, greater than 1. The cells of the square will be populated with numbers ranging from n to n-squared. Here's a magic square where n = 3.

```
Magic Square
n=3, m=15
4 9 2
3 5 7
8 1 6
```

Most of the sources I've looked at describe the method of Simon de la Loubère, or the Pyramid Method (this was the one I learned from ***The Moscow Puzzles*** by Boris A. Kordemsky). Knuth offers a different method. You start in the cell below the center cell. Then you move one square right and one down (along the left-right downwards diagonal). An important idea to grasp is that we treat the top row as if it sat below the bottom row and the left column as if it sat to the right of the right column. Therefore, if the diagonal movement would take you off the bottom of the grid, you place the number at the top of the next column. If a move takes you off to the right, you place the next number in the appropriate cell of the first column. If a cell is occupied, you place the number in the cell two cells below the last one you placed.

To make this algorithm a bit clearer, let's work through the magic square where n = 5. We need to translate this human-oriented algorithm into something the computer can follow. First, we need an array with n-squared elements, all initialised to zero. I've set this out as a grid, with the indices in bold type. We start by placing '1' in the cell below the centre of the grid (which is index 17).

0	**1**	**2**	**3**	**4**
0	0	0	0	0
5	**6**	**7**	**8**	**9**
0	0	0	0	0
10	**11**	**12**	**13**	**14**
0	0	0	0	0
15	**16**	**17**	**18**	**19**
0	0	1	0	0
20	**21**	**22**	**23**	**24**
0	0	0	0	0

The movement along the downwards left-right diagonal corresponds to adding n + 1 to the index of the current cell. The next number is therefore in cell 23. If we add 6 to 23 we get 29, which takes us off the grid. Therefore we subtract n-squared (25) which gives us 4: the correct index in which to place the number '3'. The next number goes in the first column, because we are in the right-most column at the moment. In my program, I've made this a special case as the "add n+1" rule won't work. We can then fill in the number '5' by our normal rule. After this the "add n+1" rule takes us to the square occupied by number '1'. The algorithm says we need to go two rows down from the last cell we filled, which we can achieve by adding 2n (10, in this case) to the index of the last square we filled.

[1] This is a lie. We did have fizzy drinks.

Here's the grid at the point we've reached so far:

0	**1**	**2**	**3**	**4**
0	0	0	0	3
5	**6**	**7**	**8**	**9**
4	0	0	0	0
10	**11**	**12**	**13**	**14**
0	5	0	0	0
15	**16**	**17**	**18**	**19**
0	0	1	0	0
20	**21**	**22**	**23**	**24**
0	6	0	2	0

If you continue filling in the boxes using the rules we have developed, you should find it works out. The completed grid looks like this:

0	**1**	**2**	**3**	**4**
11	24	7	20	3
5	**6**	**7**	**8**	**9**
4	12	25	7	16
10	**11**	**12**	**13**	**14**
17	5	13	21	9
15	**16**	**17**	**18**	**19**
10	18	1	14	22
20	**21**	**22**	**23**	**24**
23	6	19	2	15

Here's the algorithm we just followed:

Magic Square algorithm (for squares of order n, n is odd, n >1)

```
1. set i to int($n^2$ / 2) + 1            # this is the cell up and left from the one we start with
   set j to '1'                          # j holds the numbers we are putting in the grid
2. If j > $n^2$, we are done
   if at the end of a row:
        i2 = (i + 1) % nSq               # i2 holds the index of the cell into
   else:                                 # which we are hoping to place the current value of j.
        i2 = (i + n + 1) % nSq           # The modulus operation means we don't have to worry
3. if cell i2 is not empty:              # about going off the end of the grid.
        i2 = (i + n * 2) % nSq
put j in cell i2
set i to i2, add 1 to J
go to 2
```

It might be easier to look at the actual code! First there's a function to make sure the user only gives us an odd number greater than 1.

```
# magic_squares.py
# Knuth TAOCP, vol. 1.3.2 ex.21
from math import *

def getN():
    while 1:
        try:
            n = int(input("n: "))
            if n % 2 == 1 and n > 1:
                return n
            else:
                raise ValueError
        except ValueError:
            print("Oops")
```

Next I have a function called "makeString()" which turns the magic square into something that is actually printable:

```
def makeString(array, n, nSq):
    string = "\nMagic Square"
    string += "\nn={0}, m={1}\n".format(n, int(n * (nSq + 1) / 2))
    padding = ceil(log(nSq, 10))
    for i, cell in enumerate(array):
        if i % n == 0:
            string += '\n'
        else:
            string += ' '
        string += str(array[i]).zfill(padding)
    return string
```

I decided to pad the numbers with zeros so that they are all the same width in the program's output. To find out how many digits the largest number on the square is, I've used the expression:

```
      ceil(log(nSq, 10))
```

This means the ceiling of the base 10 logarithm of n^2. Python's int() function just gives the integer part of a number, discarding the decimal portion. The ceil() funtion is a bit more specific, ceil(x) returns x if x is an integer, otherwise it returns the next largest whole number. There's also floor() which does the inverse. It's useful to know these functions if you are dealing with negative values, for which int() may not give the result you want:

```
>>> from math import *
>>> int(2.5)
2
>>> int(-2.5)
-2
>>> floor(-2.5)
-3
>>> ceil(-2.5)
-2
```

The base 10 logarithm of x is the power to which we would need to raise 10 to get x. In the magic square where n is 5, n^2 is 25 and the ceiling of the base 10 log of 25 is 2. That means that we need to pad all the numbers to length '2' in order for the table to look neat.

You could find this out by doing len(str(nSq)), but I think the logarithm way is much cooler!

The makeString() function also tells the user the number 'm', which is the sum of the rows, columns and major diagonals.

Here's the main() function.

```
def main():
    print("*** Magic Squares ***\nPlease enter an odd number greater than one.")
    n = getN()
    normal, nSq = n + 1, n * n,
    array, i = [0] * nSq, int(nSq/2) - 1
    for j in range(1, nSq + 1):
        if (i+1) % n == 0:
            i2 = (i + 1) % nSq
        else:
            i2 = (i + normal) % nSq
        if array[i2] != 0:
            i2 = (i  + n * 2) % nSq
        array[i2] = j
        i = i2
    print(makeString(array, n, nSq))

if __name__ == "__main__":
    main()
```

Below you will see the output from the program for n = 23. I must admit I haven't checked all the columns and rows, but I'm pretty sure that the algorithm we've been looking at has given the correct result.

```
*** Magic Squares ***
Please enter an odd number greater than one.
n: 23

Magic Square
n=23, m=6095

254 519 232 497 210 475 188 453 166 431 144 409 122 387 100 365 078 343 056 321 034 299 012
013 255 520 233 498 211 476 189 454 167 432 145 410 123 388 101 366 079 344 057 322 035 277
278 014 256 521 234 499 212 477 190 455 168 433 146 411 124 389 102 367 080 345 058 300 036
037 279 015 257 522 235 500 213 478 191 456 169 434 147 412 125 390 103 368 081 323 059 301
302 038 280 016 258 523 236 501 214 479 192 457 170 435 148 413 126 391 104 346 082 324 060
061 303 039 281 017 259 524 237 502 215 480 193 458 171 436 149 414 127 369 105 347 083 325
326 062 304 040 282 018 260 525 238 503 216 481 194 459 172 437 150 392 128 370 106 348 084
085 327 063 305 041 283 019 261 526 239 504 217 482 195 460 173 415 151 393 129 371 107 349
350 086 328 064 306 042 284 020 262 527 240 505 218 483 196 438 174 416 152 394 130 372 108
109 351 087 329 065 307 043 285 021 263 528 241 506 219 461 197 439 175 417 153 395 131 373
374 110 352 088 330 066 308 044 286 022 264 529 242 484 220 462 198 440 176 418 154 396 132
133 375 111 353 089 331 067 309 045 287 023 265 507 243 485 221 463 199 441 177 419 155 397
398 134 376 112 354 090 332 068 310 046 288 001 266 508 244 486 222 464 200 442 178 420 156
157 399 135 377 113 355 091 333 069 311 024 289 002 267 509 245 487 223 465 201 443 179 421
422 158 400 136 378 114 356 092 334 047 312 025 290 003 268 510 246 488 224 466 202 444 180
181 423 159 401 137 379 115 357 070 335 048 313 026 291 004 269 511 247 489 225 467 203 445
446 182 424 160 402 138 380 093 358 071 336 049 314 027 292 005 270 512 248 490 226 468 204
205 447 183 425 161 403 116 381 094 359 072 337 050 315 028 293 006 271 513 249 491 227 469
470 206 448 184 426 139 404 117 382 095 360 073 338 051 316 029 294 007 272 514 250 492 228
229 471 207 449 162 427 140 405 118 383 096 361 074 339 052 317 030 295 008 273 515 251 493
494 230 472 185 450 163 428 141 406 119 384 097 362 075 340 053 318 031 296 009 274 516 252
253 495 208 473 186 451 164 429 142 407 120 385 098 363 076 341 054 319 032 297 010 275 517
518 231 496 209 474 187 452 165 430 143 408 121 386 099 364 077 342 055 320 033 298 011 276
```

Happy Hacking!

antiloquax