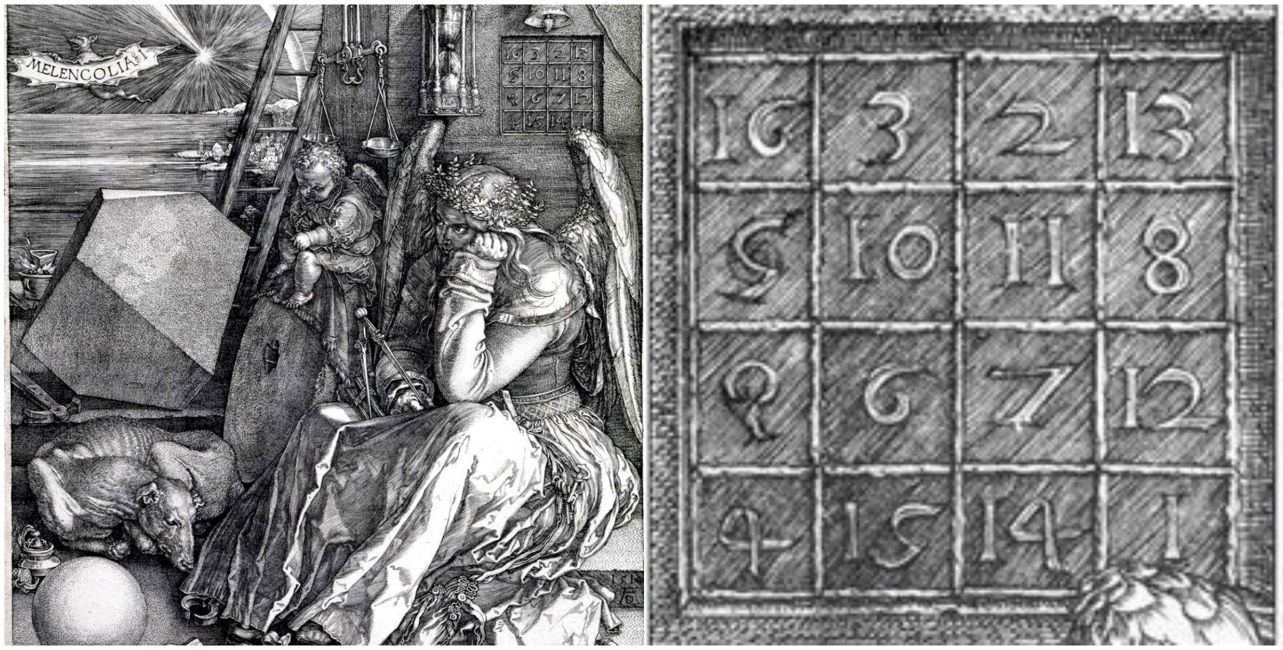
Making Magic

https://people.sc.fsu.edu/∼jburkardt/classes/math1800_2023/magic/magic.pdf



Albrecht Dürer's Melencolia II includes a magic square in the upper right corner,
He fiddled with the numbers so that the date 1514 appeared in the bottom row.

A magic square is a classic math object

- *Magic squares have intrigued recreational mathematicians for centuries;*
- *Algorithms have been found for creating squares of odd or even order;*
- *An odd algorithm is defined by Christian Hill in his textbook;*
- *To set up a square requires that we be able to do certain simple tasks;*
- *We will create a magic square using Python lists; it won't be easy!*
- *We will conclude that a better data structure is needed for numerical vectors and matrices;*

1 Magic squares

A magic square of order n is an $n \times n$ array of numbers such that all rows, all columns, and all diagonals have the same sum. (In a "semi-magic square", this is only true for the rows and columns.) It is not obvious at first how to construct a magic square, and hence they are often considered to be lucky charms with magic properties.

A magic square is a simple example of a `matrix`, that is, a rectangular array whose elements are numeric. When programmers need a "random" matrix to illustrate some procedure, they frequently use a magic matrix, since the entries are distinct integers, and the matrix has some interesting properties. MATLAB, for example, has a built-in `magic(n)` function to create a magic matrix of any order.

We can certainly use a Python list of lists to store a magic square, as, for example:

```
A = [
  [ 8, 1, 6 ],
  [ 3, 5, 7 ],
  [ 4, 9, 2 ] ]
```

But our question today is, how can we use Python to create a new magic square, by following the steps of the magic square algorithm? Assuming the algorithm is clear, we would hope that we know enough Python to get a suitable program fairly quickly and easily.

Creating a magic matrix is really a test example for us. We will soon be wanting to create and use numerical vectors and matrices, and perform all sorts of linear algebra operations with them. We really need to be comfortable with the Python tools we will use to do this.

2 An algorithm for magic matrices of odd order

I remember learning this algorithm in third grade!

To create an $n \times n$ magic square, draw a grid of empty boxes to hold your values. Number the rows from top to bottom, and the columns from left to right.

1. Start in the middle of the top row, and let $k = 1$.
2. Write k in the current grid position;
3. If $k = n^2$, the grid is complete, so stop;
4. Else, set $k = k + 1$;
5. Plan to move diagonally up and right. But if this move leaves the grid, wrap to the first column or last row.
6. If this cell is already filled, move vertically down one space instead;
7. Return to step 2.

While the algorithm seems to make sense, let's work through some examples to see what is being talked about! Create a suitable empty grid of cells, and then recreate the magic matrices of order 3, 5, and 7.

The 3×3 example:

$$A_3 = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

The 5×5 example:

$$A_5 = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix}$$

The 7×7 example:

$$A_7 = \begin{bmatrix} 30 & 39 & 48 & 1 & 10 & 19 & 28 \\ 38 & 47 & 7 & 9 & 18 & 27 & 29 \\ 46 & 6 & 8 & 17 & 26 & 35 & 37 \\ 5 & 14 & 16 & 25 & 34 & 36 & 45 \\ 13 & 15 & 24 & 33 & 42 & 44 & 4 \\ 21 & 23 & 32 & 41 & 43 & 3 & 12 \\ 22 & 31 & 40 & 49 & 2 & 11 & 20 \end{bmatrix}$$

3 Implementing the algorithm

There are several tasks we need to do computationally, in order to carry out this algorithm in Python.

We **need to**:

- create, in advance, a place to store the entries, that is a list of n rows, each row a list of n values.
- specify any entry using indexing.
- tell when a planned move takes us out of the grid.

We would also **like to**:

- easily print the resulting magic matrix.
- easily compute the row, column and diagonal sums.
- compute the row and column sums using matrix multiplication.

This very last item can best be illustrated by using MATLAB. If A_n is a magic matrix of order n, and v is a vector of 1's of length n, then $A * x$ is the vector of row sums, and $A^T * x$ is the vector of column sums. There are other commands that allow us to also check the diagonal and antidiagonal sums.

```
%  Warning: This is MATLAB code, not Python!
n = 7
Asum = ( n * ( n^2 + 1 ) ) / 2
A = magic ( n )
x = ones ( n, 1 )
A*x
A'*x               % Transpose of A times x
sum ( diag(A) )
sum ( diag ( flipud(A) )) % Flip A upside down to get antidiagonal
```

Although we could presumably check magic squares in other ways, we will really wish to be able to use more sophisticated tools for later numerical calculations.

4 Allocating an $n \times n$ list of lists

We can create an empty list by the statement

```
x = []
```

We can create a list of 3 numbers, or a list of lists of 9 numbers by

```
y = [ 1, 2, 3]
A = [ [ 1, 2, 3], [ 4, 5, 6], [7, 8, 9] ]
```

These statements assume I already know the size and contents of the lists.

What if we don't know the size of a list in advance, but we need to create it and initialize it with n zeros, where the value of n is not known beforehand? I claimed earlier that we could write a (rather ugly) expression

```
z = [0] * n
```

But none of this will help us if we want to create an $n \times n$ list of lists. What can we do?

Well, we can certainly set up one row of zeros of length n. We can start with an empty list and then simply append another zero n times:

```
row = []
for i in range ( 0, n ):
  row.append ( 0 )
```

and if we print the result with $n = 5$ say, we get what we expect.

Now it almost seems like we can repeat this trick to create our matrix. We simply let A be an empty list, and append n copies of `row`, something like this:

```
A = []
for i in range ( 0, n ):
  A.append ( row )
```

and indeed, if we now print A, it does look like a somewhat rathe raggedy version of a matrix.

But there's a terrible surprise waiting for us. I only found it when I was knee-deep in the magic matrix algorithm itself. But let's try to catch it early. Suppose we set the middle entry of row 0 to 1, like the algorithm tells us to do. Remember that we use a pair of bracketed index values to access a specific entry of a list of lists, so we write

```
A[0][2] = 1
```

Now we print out the matrix, but to our surprise, every cell in column 2 has been set to 1. If we then try to set the next value,

```
A[4][2] = 2
```

we find that every entry in column 3 has been set to 2. What madness is going on?

Remember that I said earlier that Python has a peculiar understanding of the equals sign. If we write something like `object2 = object1`, then sometimes Python does not copy the value of object1 into object2. Instead, it understands us to want object2 to be another name for the same single set of values.

What is happening to our A list of lists is that every row has been set to be another name for the original `row` data item. To prove this, let's print `row`, which presumably should still be entirely zero. Instead, we see this:

```
[ 0, 0, 1, 2, 0 ]
```

The new values in `row` were put there by our assignment statements to `A`, and then got displayed repeatedly when we printed `A`.

While the problem is deceptive and easy to overlook at first, the fix is not difficult. Whenever you encounter such an ambiguous situation, you have to use the `.copy()` method to make it clear that you want to create a new object with a new copy of the values of the old object. In other words, we create A as follows:

```
A = []
for i in range ( 0, n ):
  A.append ( row.copy() )
```

and now if you try to assign a few values to entries of `A`, the right things happen.

5 Implementing the algorithm

Assuming we have set up space for the list of lists that will hold A, the remainder of our Python implementation is:

```
  k = 1
  i = 0
  j = n // 2

  while ( k <= n**2 ):
    A[i][j] = k
    k = k + 1

    new_i = ( i - 1 ) % n  # Automatically wraps around if necessary
    new_j = ( j + 1 ) % n
#
#  The next statement is True if A[new_i][new_j] is not zero.
#  In  that case, we need to drop down 1.
#
    if ( A[new_i][new_j] ):
      i = i + 1
    else:
      i = new_i
      j = new_j

  return A
```

6 Complaints

Now we have managed to create a magic matrix of arbitrary odd order n, using the Python `list` datatype. But I would say it was not a happy experience, especially if we plan to do linear algebra work.

Because we did not know the value of n in advance, our algorithm had to do some awkward and unfamiliar things in order to set up and zero out the $n \times n$ grid.

The `sum()` function is not able to be applied in a simple way to the rows, columns and diagonals of `A` in order to confirm that it is a magic matrix.

The `print()` command does a bad job of showing our list of lists as a matrix. In that case, however, there is a solution, called *prettyprint*:

```
  import pprint

  pp = pprint.PrettyPrinter ( indent = 2 )
  pp.pprint ( A )
```

As we mentioned earlier, mathematically, we could use matrix multiplication to verify built in matrix-vector multiplication, or matrix transpose, and don't even think about matrix inverse, determinant, or eigenvalues!

While the Python `list` datatype is flexible, efficient and powerful for processing nonnumerical data, it is insufficient for computations involving numerical data. This is the reason for the creation of a whole new data library called `numpy`. We will have to learn some new ways of working, but we will be much more comfortable for many of the most important tasks in scientific work.

7 The numpy alternative

Just to give you an idea of what `numpy` looks like, here is the rewritten magic matrix algorithm:

```
import numpy as np
n  = 5
A = np.zeros ( ( n, n ), dtype = int )

k = 1
i = 0
j = n // 2

while k <= n**2:
  A[i,j] = k
  k = k + 1
  newi = ( i - 1 ) % n
  newj = ( j + 1 ) % n
  if A[newi,newj]:
    i = i + 1
  else:
    i = newi
    j = newj

print ( A )
```