# crossnumber Documentation

**Adam Vellender**

**Aug 24, 2021**

# CONTENTS:

Welcome to the documentation for the *crossnumber* Python3 package.

CHAPTER

ONE

# WHAT IS "CROSSNUMBER"?

Crossnumber is a collection of functions for the Python3 programming language that can be useful in helping solve crossnumbers. The functions are fairly simple but can speed up solving by making it easier to generate lists of special numbers, check factors, digit sums, and so on.

If you're not familiar with Python, it's a programming language that places emphasis on ease of learning; the syntax is straightforward. I'd recommend Codecademy as a good beginner's tutorial although there are very many others available online.

It probably goes without saying that the main objective in solving puzzles should be enjoyment, so computer use should probably be fairly minimal. Brute-forcing your way to a puzzle's solution is not fun, but then nor perhaps is manually looking through a list of four-digit prime numbers to find ones with an even first digit and a third digit that's either 8 or 9! The functions in *crossnumber*, used sparingly, can help with some of that kind of heavy lifting.

CHAPTER

TWO

# INSTALLATION

You'll need a working Python installation. If you've never used Python, installing Anaconda might be the easiest way on most platforms.

The *crossnumber* Python package can then be installed using the `pip` package manager:

```
pip install crossnumber
```

If you're new to Python and unfamiliar with pip, search online for how to use it, although assuming you have Python installed, it should be as simple as typing the above into a command prompt.

CHAPTER

# THREE

# USAGE

To import all of *crossnumber*'s functions into your Python session, run

```
from crossnumber import *
```

Perhaps it's worth noting here that importing all functions from a package like this (using an asterisk) is generally a bad idea, but for something non-vital like solving a crossnumber it's fine.

After this command has been executed, you'll be able to use all of the *crossnumber* functions.

A complete list of functions is available in the *list of functions* near the end of this page, but this section of the documentation is intended to give you an idea of how you can use them.

## 3.1 Generating special numbers

Let's start with the basics. When solving crossnumbers you quite often need to refer to a list of a certain type of number of certain length. For instance, primes of length 2, triangular numbers of length 3, Fibonacci numbers of length 4, and so on. The *crossnumber* package has functions to generate these, for instance:

```
pol(2)
```

will output a tuple containing the primes of length 2:

```
(11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97)
```

The `pol` here stands for *primes of length*. Other functions to generate special types of number of certain length include:

- `sol(n)` for square numbers of length `n`;
- `tol(n)` for triangular numbers of length `n`;
- `fol(n)` for Fibonacci numbers of length `n`;
- `col(n)` for cube numbers of length `n`.
- `iol(n)` for all integers of length `n`.

These can be simply combined; if you needed a list of all primes under 1000 for instance, `pol(1)+pol(2)+pol(3)` would do the job.

## 3.2 Applying simple constraints

Suppose you're solving a puzzle where you have a four-digit prime whose second digit you know to be 4 and third digit you know to be 7. You can simply ask Python to run through all primes of length 4 and print just those matching those conditions:

```
for a in pol(4):
   if nthDigit(a,2)==4 and nthDigit(a,3)==7:
      print(a)
```

which will output:

```
1471
2473
2477
5471
5477
5479
6473
7477
9473
9479
```

Here we've used the *crossnumber* function `nthDigit(a,b)` which returns the `b`-th digit of the integer `a` (e.g. `nthDigit(42,2)` will return `2`).

As another example, suppose we're looking for a three-digit Fibonacci number whose middle digit is the final digit of a three-digit square. We could get a list of such pairs via

```
for a in fol(3):
   for b in sol(3):
      if nthDigit(a,2)==nthDigit(b,3):
         print(a,b)
```

Entirely equivalently, there's a `match` function, which can save a little bit of typing, with syntax like this:

```
for a in fol(3):
   for b in sol(3):
      if match(a,2,b,3):
         print(a,b)
```

Incidentally, the output for either of the above blocks of code looks like this:

```
144 144
144 324
144 484
144 784
610 121
610 361
610 441
610 841
610 961
```

Of course, in real puzzles, you often have partial information about a digit. For instance, suppose we're looking for a four-digit triangular number with second digit either 2 or 5, and third digit 7. We'll just have to adjust our for-loop to deal with such cases:

```
for a in tol(4):
   if nthDigit(a,2) in [2,5] and nthDigit(a,3)==7:
     print(a)
```

which will output:

```
1275
2278
3570
4278
```

Other things that might be worth mentioning here, especially for those not too familiar with Python, include that you can check divisibility easily. Suppose you're looking for a two-digit integer that's triangular but not a multiple of 3. Python can give you the remainder of an integer, `a` say, when divided by 3 via the command `a%3`. If `a` is not a multiple of 3, then this will be not equal to 0 (in Python's syntax: `a%3 != 0`). Thus

```
for a in tol(2):
   if a%3 != 0:
      print(a)
```

will do the described search for you.

Finally, perhaps it's worth mentioning that you can use `not in`, as well as just `in`, in searches.

## 3.3 Digit sum and product, anagrams, and reversals.

Common types of clue in crossnumbers include using things like digit sums, digit products, anagrams/jumbles, reversals of integers, and so on. There are functions in *crossnumber* for all of these.

- `digitSum(n)` will return the digit sum of `n`;
- `digitProduct(n)` will return the digit product of `n`;
- `isAnagram(m,n)` will return `True` if the integers `m` and `n` are anagrams of each other and `False` if not;
- `rev(n)` will return the reverse of an integer, e.g. `rev(123)` will return `321`.

As an example, suppose we're looking for a three-digit prime with middle digit 3 and digit sum 11:

```
for a in pol(3):
   if nthDigit(a,2)==3 and digitSum(a)==11:
      print(a)
```

The above returns one possibility: `137`.

Another example: suppose we're looking for a four-digit prime number whose reverse is a four-digit triangular number. Then

```
for a in pol(4):
   if rev(a) in tol(4):
      print(a)
```

will do the job for us, giving us four candidates:

```
1171
1801
5563
8731
```

As a final example in this section, suppose we were looking for two three-digit numbers, one square and one prime, that are anagrams of each other. Perhaps we know that the square ends in 4. The following should give us all possible candidate pairs:

```
for a in sol(3):
   for b in pol(3):
      if isAnagram(a,b) and nthDigit(a,3)==4:
         print(a,b)
```

As luck would have it, there's only one such possibility:

```
784 487
```

so our square is 784 and our prime is 487.

## 3.4 Factors

Some puzzles require you to use information about factors, or the number of factors, or prime factors, of an integer. There are many nice mathematical results concerning these, but sometimes it's useful to be able to use Python to help.

- The function `factors` outputs a tuple of all factors of an integer (including 1 and the integer itself). For instance, `factors(24)` will return `(1, 2, 3, 4, 6, 8, 12, 24)`.
- The function `pf` outputs a tuple of all *prime* factors of an integer. For instance, `pf(24)` will return `(2, 3)`.
- The function `primeFactorisation` outputs a Python dictionary, each entry of which is of the format `factor: exponent`. For instance, `primeFactorisation(24)` will return `{2: 3, 3: 1}`.

If you want to count how many factors an integer `n` has, just use Python's `len` command. As an example, suppose you're looking for a three-digit triangular number with exactly six distinct factors. Then

```
for a in tol(3):
   if len(factors(a))==6:
      print(a, factors(a))
```

will output

```
153 (1, 3, 9, 17, 51, 153)
171 (1, 3, 9, 19, 57, 171)
325 (1, 5, 13, 25, 65, 325)
```

Here, I've used `print(a, factors(a))` to tell Python to print not only the triangular numbers, but also the list of factors.

## 3.5 Annotation

It's frustrating when you realise you've made a mistake when solving a crossnumber. You've got to retrace your steps and figure out where the error was. If you're not making some sort of set of notes, this can be tortuous and you end up being back at square one. For this reason, `crossnumber` includes a few note-taking functions, so that chunks of code can be signposted with what you're investigating, what you conclude, and so on.

You can use `note("Your text here")` to make notes. This will output `***NOTE***: Your text here` in red.

Similarly, there are functions `consider`, `assumptions`, `digits`, and `conclusion` that do the same. I find them useful sometimes e.g. I might note `consider("12 across")`, and if I conclude the final digit is 3, I'd then write `digits("Final digit of 12A is 3")`.

CHAPTER

FOUR

# LIST OF FUNCTIONS

Here is a comprehensive list of functions in *crossnumber*:

Table 1: Crossnumber functions list

| Function | Description |
|---|---|
| `primes(n)` | Returns tuple of all primes strictly less than `n` |
| `pol(n)` | Returns tuple of all primes of length `n` |
| `sol(n)` | Returns tuple of all squares of length `n` |
| `col(n)` | Returns tuple of all cubes of length `n` |
| `tol(n)` | Returns tuple of all triangular numbers of length `n` |
| `iol(n)` | Returns tuple of all integers of length `n` |
| `fibonacci(n)` | Returns tuple of all Fibonacci numbers strictly less than `n` |
| `fol(n)` | Returns tuple of all Fibonacci numbers of length `n` |
| `friendlyol(n)` | Returns tuple of all 'friendly' numbers (integers divisible by their digit sum) |
| `digitSum(n)` | Returns sum of the digits in `n` |
| `digitProduct(n)` | Returns product of the digits in `n` |
| `isAnagram(m,n)` | Returns `True` if `m` and `n` are anagrams of each other and `False` otherwise |
| `nthDigit(n,d)` | Returns the *d*-th digit of *n* |
| `match(n,a,m,b)` | Returns `True` is the `a`-th digit of `n` is the same as the `b`-th digit of `m` and `False` otherwise |
| `isLength(n,l)` | Returns `True` if the integer `n` is of length `l` and `False` otherwise |
| `rev(n)` | Returns the digit reversal of `n` |
| `isFriendly(n)` | Returns `True` if the integer `n` is 'friendly' (see description of `fol`) and `False` otherwise |
| `lcm(x,y)` | Returns the lowest common multiple of `x` and `y` |
| `gcf(x,y)` | Returns the greatest common factor of `x` and `y` |
| `factors(n)` | Returns a tuple of the factors of `n`, including 1 and `n` |
| `pf(n)` | Returns a tuple of the distinct prime factors of `n` |
| `primeFactorisation(n)` | Returns a dictionary detailing the prime factorisation of `n`. Each entry is of the format `factor: exponent` |
| `note('Message')` | Note-taking function, outputs `***NOTE***: Message` to the console |
| `conclusion('Message')` | Note-taking function, outputs `***CONCLUSION***: Message` to the console |
| `assumption('Message')` | Note-taking function, outputs `***ASSUMPTION***: Message` to the console |
| `consider('Message')` | Note-taking function, outputs `***CONSIDER***: Message` to the console |
| `digits('Message')` | Note-taking function, outputs `***DIGITS***: Message` to the console |