Back to Python

Python is quite a powerful language! Let's find out how by looking at an interesting example proposed by Tucker Balch.

You've almost reached the end of the course and, hopefully, you've learned that NumPy is a very versatile and powerful library. However in the meantime, remember that Python is also quite a powerful language. In fact, in some specific cases, it might be more powerful than NumPy.

Allocations

Let's consider, for example, an interesting exercise that has been proposed by Tucker Balch in his Computational Investing course. The exercise is written as:

#

> *Write the most succinct code possible to compute all "legal" allocations to 4 stocks such that the allocations are in 1.0 chunks, and the allocations sum to 10.0.*

Yaser Martinez collected the different answers from the community and the proposed solutions yield surprising results. But let's start with the most obvious Python solution:

main.py

tools.py

This solution is the slowest solution because it requires 4 loops, and more importantly, it tests all the different combinations (14641) of 4 integers between 0 and 10 to retain only combinations whose sum is 10. We can, of course, get rid of the 4 loops using itertools, but the code remains slow:

main.py

tools.py

```
import itertools as it
from tools import timeit

def solution_2():
  # Itertools
  # 14641 (=11*11*11*11) iterations & tests
  return [(i,j,k,l)
          for i,j,k,l in it.product(range(11),repeat=4) if i+j+k+l == 10]

timeit("solution_2()", globals())
```

One of the best solution that has been proposed by Nick Popplas takes advantage of the fact we can have intelligent imbricated loops that will allow us to directly build each tuple without any test as shown below:

main.py

tools.py

```
from tools import timeit

def solution_3():
  return [(a, b, c, (10 - a - b - c))
          for a in range(11)
          for b in range(11 - a)
          for c in range(11 - a - b)]

timeit("solution_3()", globals())
```

The best NumPy solution by Yaser Martinez uses a different strategy with a restricted set of tests:

main.py

tools.py

```
from tools import timeit
import numpy as np


def solution_4():
  X123 = np.indices((11,11,11)).reshape(3,11*11*11)
  X4 = 10 - X123.sum(axis=0)
  return np.vstack((X123, X4)).T[X4 > -1]

timeit("solution_4()", globals())
```

If we benchmark these methods, we get:

main.py

tools.py

```
# -----------------------------------------------------------------------------
# From Numpy to Python
# Copyright (2017) Nicolas P. Rougier - BSD license
# More information at https://github.com/rougier/numpy-book
# -----------------------------------------------------------------------------
import numpy as np
import itertools as it


def solution_1():
    # Author: Tucker Balch
```

```
    # Brute force
    # 14641 (=11*11*11*11) iterations & tests
    Z = []

    for i in range(11):
        for j in range(11):
            for k in range(11):
                for l in range(11):
                    if i+j+k+l == 10:
                        Z.append((i,j,k,l))
    return Z


def solution_2():
    # Author: Daniel Vinegrad
    # Itertools
    # 14641 (=11*11*11*11) iterations & tests
    return [(i,j,k,l)
            for i,j,k,l in it.product(range(11),repeat=4) if i+j+k+l == 10]

def solution_3():
    # Author: Nick Poplas
    # Intricated iterations
    # 486 iterations, no test
    return [(a, b, c, (10 - a - b - c))
            for a in range(11) for b in range(11 - a) for c in range(11 - a - b)]


def solution_3_bis():
    # Iterator using intricated iterations
    # 486 iterations, no test
    return ((a, b, c, (10 - a - b - c))
            for a in range(11) for b in range(11 - a) for c in range(11 - a - b))


def solution_4():
    # Author: Yaser Martinez
    # Numpy indices
    # No iterations, 1331 (= 11*11*11) tests
    X123 = np.indices((11,11,11)).reshape(3,11*11*11)
    X4 = 10 - X123.sum(axis=0)
    return np.vstack((X123, X4)).T[X4 > -1]


if __name__ == '__main__':
    from tools import timeit

    timeit("solution_1()", globals())
    timeit("solution_2()", globals())
    timeit("solution_3()", globals())
    timeit("solution_4()", globals())
```

The NumPy solution is the fastest but the pure Python solution is comparable. But let me introduce a small modification to the Python solution. Let’s benchmark this modification and see what we get:

main.py

tools.py

```python
from tools import timeit


def solution_3_bis():
  return ((a, b, c, (10 - a - b - c))
          for a in range(11)
          for b in range(11 - a)
          for c in range(11 - a - b))

timeit("solution_3_bis()", globals())
```

You read that right, we have gained a factor of 100 just by replacing square brackets with parenthesis. How is that possible? The explanation can be found by looking at the type of the returned object:

```python
def solution_3():
    # Author: Nick Poplas
    # Intricated iterations
    # 486 iterations, no test
    return [(a, b, c, (10 - a - b - c))
            for a in range(11) for b in range(11 - a) for c in range(11 - a - b)]


def solution_3_bis():
    # Iterator using intricated iterations
    # 486 iterations, no test
    return ((a, b, c, (10 - a - b - c))
            for a in range(11) for b in range(11 - a) for c in range(11 - a - b))

print(type(solution_3()))
# <class 'list'>
print(type(solution_3_bis()))
# <class 'generator'>
```

The `solution_3_bis()` returns a generator that can be used to generate the full list or to iterate over all the different elements. In any case, the huge speedup comes from the non-instantiation of the full list and it is thus important to wonder if you need an actual instance of your result or if a simple generator might do the job.

In the next lesson, we will look at a bunch of other useful Python packages and their uses!