NumPy

Janusz Jabłonowski, MIMUW

Windows problem

- RuntimeError: The current Numpy installation (<path>) fails to pass a sanity check due to a bug in the windows runtime.
- Way around [Dec. 2020]: pip install numpy==1.19.3

Timing snippets of code

- The module `timeit`
- Can be used as a function or as a program (from command line)
- There are other possibilities but this is extremely handy
- There are several functions in this module, we need just one

timeit

- Syntax

```
timeit.timeit(stmt='pass', setup='pass', timer=<default timer>,
number=1000000, globals=None)
```

- Parameters
  - stmt - the snippet to be measured. One line or several lines with line breaks
    - “”” “”” or “\n” inserted manually or ; for simple statements
  - setup - the initialization, executed only once, before start of the tests
  - timer - skipped here (`time.perf_counter`)¶
  - number - number of test repetitions
  - globals - environment with nonlocal variables used by the test
- Result
  - the total time of *number* executions of the *stmt* (time of setup is not counted) in seconds

Simple example of timeit

```
>>> timeit.timeit(stmt="x=x+1", setup="x=0")

0.024219600000833452

>>> timeit.timeit(stmt="x+=1", setup="x=0")

0.025806000000557106
```

- Slightly surprising

Passing nonlocal variables

- At least three methods
- Initialization in code
  - `setup=f"n = {n}; lst = {lst}"`
- Passing locals()/globals() as globals
  - `globals=locals()`
- Passing custom dictionary with needed variables
  - `globals={"n": n, "lst": lst}`

Passing nonlocal variables - who is the winner?

```
stmt =  "sum = 0" "\n" \
        "for i in range(n):" "\n" \
        "   sum+= lst[i]"

time = timeit.timeit(setup=f"n = {n}; lst = {lst}", stmt=stmt)
print(f"Initialization in the setup string:            {time}")

time = timeit.timeit(stmt=stmt,globals=locals())
print(f"Passing locals() as the global parameter:      {time}")

time = timeit.timeit(stmt=stmt, globals={"n": n, "lst": lst})
print(f"Creating a dictionary and passing as globals:  {time}")
```

Passing nonlocal variables - who is the winner?

```
Initialization in the setup string:         0.37243479999999995
Passing locals() as the global parameter:   0.4038794000000001
Creating a dictionary and passing as globals: 0.43651209999999985
```

- Times are quite similar
- Almost always the first method is the fastest

A short note

Passing code as a string

- So flexible!
- So powerful!
- So dangerous!

To avoid misunderstandings

- array != array
- array.array
  - standard library
  - one-dimensional arrays
  - efficient representation of numeric values
    - char, wchar, int, long, float, double
  - the type of array elements has to be specified by array creation
- numpy.ndarray has an alias numpy.array
  - this will be discussed on these slides

array.array (this is not numpy yet!)

- Creating arrays:

```
array.array(<type-character>, <initializer>)
```

- <type-character> one of 13 defined (‘i’ for signed int, ‘l’ for signed long)
- <initializer> has to be a list or iterable (or bytes-like object)
- The Array module provides some functions (like reverse)
- Elements are accessible through indexing

array.array speed vs lists

```
stmt =  "sum = 0" "\n" \
        "for i in range(n//2):" "\n" \
        "   lst[2*i] += lst[2*i-1]"
lst = [i for i in range(n)]
tab_int = array.array('i', lst)
tab_long = array.array('l', lst)

time = timeit.timeit(setup=f"n = {n}; lst = {lst}", stmt=stmt)
print(f"Addition on a list:                 {time}")
time = timeit.timeit(setup=f"from array import array; n={n};
lst={tab_int}", stmt=stmt)
print(f"Addition with an array of ints:     {time}")
# ... analogically for long integers
```

array.array speed vs lists - results

```
Addition on a list:                0.5396622
Addition with an array of ints:      0.8573933
Addition with an array of long ints: 0.8826565
```

Slightly disappointing ...

NumPy

NumPy

- NUMerical PYthon
- One of the most fundamental packages for numerical computations
- Especially important are arrays
- By convention it is imported as "`import numpy as np`"
- Current version 1.19 (Dec. 2020)

NumPy

NumPy provides:

- `ndarray` - fast array-oriented operations
- Fast mathematical functions on entire arrays
- Reading/writing array data to/from files
- Memory-mapped files
- Random number generators
- Linear algebra operations
- FFT
- API for using C/C++/Fortran libraries

Virtues of NumPy arrays

NumPy

- Store data internally as continuous blocks of memory
- This enables C functions to operate without a need for converting data or dynamic type-checking
- It also makes the arrays smaller than their Python-list equivalents
- NumPy provides functions on arrays which enable computations without the need for Python loops

Speed comparison

```
time = timeit.timeit(setup=f"n = {n}; lst = {list(range(n))}",
                     stmt="res = [2*i for i in lst]")
print(f"Multiplication in a list:                  {time}")

time = timeit.timeit(setup=f"import array; n = {n};"
                           "arr_arr = array.array('i', range({n}))",
                     stmt="res = arr_arr * 2")
print(f"Multiplication in an array.array:          {time}")

time = timeit.timeit(setup=f"import numpy as np; n = {n};"
                           "np_arr = np.arange({n})",
                     stmt="res = np_arr * 2")
print(f"Multiplication in an NumPy array:          {time}")
```

Speed comparison

- Results(for n=1000 and n=100 000)

```
Multiplication in a list:              0.35497769999999995
Multiplication in an array.array:      0.0023769999999999625
Multiplication in an NumPy array:      0.012580600000000053

Multiplication in a list:              38.5393915
Multiplication in an array.array:      0.2486524999999986
Multiplication in an NumPy array:      0.5705263000000045
```

- Unfortunately (for arrays) here the result is duplication of the array (still amazingly fast)
- NumPy is generally (not only here) 10 to 100 faster than its Python loop equivalent

NumPy arrays

- For efficient handling of ...
- n-dimensional (n>=1)
- homogenous (all elements of the same type, usually numbers)
- ... arrays.
- Indexed by tuples (of non-negative integers)
- Dimensions are here called *axes*
- Syntax similar (but different) to lists
- For example:

```
[[1., 2., 3.],
 [4., 5., 6.]]
```

- Has 2 axes, first of the length 2, the second of 3.

NumPy arrays - creating them

- Creating from iterables
  - `np.array(range(5))` -> [0 1 2 3 4]
  - `np.array(list(range(5)))` -> [0 1 2 3 4]
  - `np.array(n_arr)` -> [0 1 2 3 4] (assuming n_arr is [0 1 2 3 4]
  - `np.array([[i]*2]*3 for i in range(3)])`

    ->[[0 0] [1 1] [2 2]]

- The general form

  - numpy.array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0)
  - we skip the discussion of the other parameters here

NumPy arrays - creating them

- Creating with special values
- In general

```
numpy.zeros(shape, dtype=float, order='C')
numpy.ones(shape, dtype=None, order='C')
numpy.full(shape, fill_value, dtype=None, order='C')
numpy.empty(shape, dtype=float, order='C')
```

- `shape` is int or a tuple or a sequence of ints
  - the documentation in some places states that shape should be int or a tuple, but lists work as well
- `dtype` the desired type of elements of the new array
  - for ones the default is numpy.float64
  - for full None means `np.array(fill_value).dtype`
- `order` C (row-major) or Fortran (column-major) like (skip it)
- Note that empty is (and cannot be) empty! We do not care/know what is inside!

NumPy arrays - creating them

- There are also functions creating an array with shape taken from another array:
  - zeros_like, ones_like, full_like, empty_like
  - We do not discuss them here
- Examples of creating with special values
  - `np.zeros(3)` -> [0. 0. 0.]
  - `np.ones((1,2,3))` -> [[[1. 1. 1.] [1. 1. 1.]]]
  - `np.full([2,3],13)` -> [[13 13 13] [13 13 13]]

NumPy arrays - creating them

There are some special arrays that can be easily created

- numpy.identity(n, dtype=None)
  - square, 2-D one
- numpy.eye(N, M=None, k=0, dtype=<class 'float'>, order='C')
  - rectangular N*M array with ones on th k-th diagonal and zeros everywhere
- numpy.arange([start,]stop, [step,]dtype=None)
  - like regular range, but here floats are also allowed
  - np.arange(0.5, 1.5, 0.1) -> [0.5 0.6 0.7 0.8 0.9 1. 1.1 1.2 1.3 1.4]

NumPy arrays - creating them

- Creation with random numbers
  - `np.arange(n)` - linear array
  - `np.random.randn(4, 2)` - two dimensional, randomly filled
  - `np.random.randn(2, 3, 4)` - three dimensional
-

NumPy arrays

- Asking for information:
  - shape (returns a tuple with sizes)
  - `np.random.randn(2,3,4).shape` -> `(2, 3, 4)`
  - dtype (type - one only - of data within)
  - `np.random.randn(2,3,4).dtype` -> `dtype('float64')`
  - ndim - number of dimensions
  - `np.random.randn(2,3,4).ndim` -> `3`

The type of an array elements

- Numpy can guess the type of elements during creation

```
tab1 = np.array([1,2,3])          -> int32
tab2 = np.array([1., 2., 3.]) -> float64
```

- Or it may infer it from types of array arguments

```
tab3 = tab1+tab2                  -> float64
```

NumPy arrays element types

- Different than in Python
- Chosen for efficiency and compatibility with oter programming languages (C, Fortran)
- Currently 19 of them
- Most interesting ones:
  - float64
  - int64, int32 (signed)
  - complex64
  - bool
  - object (any Python object)
  - string_ (fixed-length ASCII string)
  - unicode_ (fixed length unicode string, number of bytes per character is platform specific)

NumPy arrays element types

- They may be casted with astype

```
tab = np.array([1,2,3])
print(f"{tab=}, {tab.dtype=}")
tab = tab.astype(np.float64)
print(f"{tab=}, {tab.dtype=}")
```

- Result:

```
tab=array([1, 2, 3]), tab.dtype=dtype('int32')
tab=array([1., 2., 3.]), tab.dtype=dtype('float64')
```

NumPy arrays element types

- General form

```
ndarray.astype(dtype, order='K', casting='unsafe', subok=True,
copy=True)
```

- Allows even for
  - truncating (floats to ints)
  - converting (strings to numbers)
- Always creates a new array (even if the type is the same)
- Throws the exception ValueError if casting fails

NumPy arrays - vectorization

Vectorization

- Any arithmetic operation on arrays of the same size is applied element-wise
- Array arithmetic: array@array where @ stands for any arithmetic operator like +, -, *, /, **
  - understood as item by item operations!

```
tab1 = np.array([1,2,3])
tab2 = np.array([2,3,4])
print(tab1**tab2)
```

```
[ 1  8 81]
```

NumPy arrays

- Operations on scalars are allowed as well operations (tab@scalar, scalar@tab) where @ stands for +,-,*,/, ...
- They are applied to each array element (with the same scalar)

NumPy arrays - comparisons

- Comparisons are allowed as well
- And also applied element by element
- They create new arrays

```
tab1 = np.array([1,2,3])
tab2 = np.array([2,3,4])
print(tab1 == 3)
print(tab1 > tab2)
print(tab1+1 == tab2)
[False False  True]
[False False False]
[ True  True  True]
```

NumPy arrays - indexing

- Indexing works like in Python (but more options are available, see further)

```
tab = np.full([2, 3],7)
print(tab)
tab[1][2] = 8
print(tab)
tab[1]=[1,2,3]
print(tab)
print(tab[-1])
[[7 7 7] [7 7 7]]
[[7 7 7] [7 7 8]]
[[7 7 7] [1 2 3]]
[1 2 3]
```

NumPy arrays - slicing

- Slicing has more options
- It creates a view, never a copy!
    - rationale: efficiency!
    - For copy use .copy()

```
tab  = np.arange(8)
tab2 = tab[2:4]
print(f"{tab=} {tab2=}")
tab2[1] = 9
print(f"{tab=} {tab2=}")
```

```
tab=array([0, 1, 2, 3, 4, 5, 6, 7]) tab2=array([2, 3])
tab=array([0, 1, 2, 9, 4, 5, 6, 7]) tab2=array([2, 9])
```

NumPy arrays - slicing

- Assignments to slices are also possible (as in Python)

```
tab, tab2  = np.arange(8), tab[2:4]
print(f"{tab=} {tab2=}")
tab2[1] = 9
print(f"{tab=} {tab2=}")
tab[2:4] = [1,2]
print(f"{tab=} {tab2=}")
```

```
tab=array([0, 1, 2, 3, 4, 5, 6, 7]) tab2=array([2, 3])
tab=array([0, 1, 2, 9, 4, 5, 6, 7]) tab2=array([2, 9])
tab=array([0, 1, 1, 2, 4, 5, 6, 7]) tab2=array([1, 2])
```

NumPy Indexing with tuples

- Reminder: in Python 1, 2 (or even 1,) stands for a tuple: (1,2) (1,)
- NumPy allows for tuples as indices

```
tab = np.full((2,3),7)
print(f"{tab[1]=}, {tab[1][2]=} {tab[1,2]=}")

tab[1]=array([7, 7, 7]), tab[1][2]=7 tab[1,2]=7
```

NumPy Indexing with tuples

- Of course some of the last indices may be omitted

```
tab= np.array([[[i for i in range(j*k, j*k+2)] for j in
     range (k, k+2)] for k in range(1,3)])
print(f"{tab[0]=}, {tab[0,1]=}")")

tab[0]=array([[1, 2], [2, 3]]), tab[0,1]=array([2, 3])
```

- The result is a **view** of some subarray

NumPy Indexing with slices again

- Slices for multidimensional arrays are quite more sophisticated
- Can be mixed with indexes
- Return **views**
- See the examples on the next page

NumPy Indexing with slices again

```
tab= np.array([[i for i in range(j*3, j*3+3)]
                            for j in range (0, 3)])
print(f"{tab=}")
print(f"{tab[1:,1:]=}\n{tab[:,0]=}\n{tab[2,1:]=}")
```

```
tab=array([[0, 1, 2],
           [3, 4, 5],
           [6, 7, 8]])
tab[1:,1:]=array([[4, 5], [7, 8]])
tab[:,0]=array([0, 3, 6])
tab[2,1:]=array([7, 8]))
```

Functions working on entire arrays

- Mathematical functions applied to every element
- Return another array
    - sqrt - square root
    - cbrt - cubic root
    - square - square
    - absolute, abs, fabs - absolute value
    - log, log10, log2
    - exp
    - sign
    - floor, cell
    - sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, arctanh
    - ...

Functions working on entire arrays

- Mathematical functions applied to every pair of elements of two arrays
- Return another array
    - add, subtract, multiply, divide, floor_divide.
    - power
    - maximum, minimum, fmax, fmin (f versions ignore NaN)
    - comparisons (greater, greater_equal, ...)
    - ...

Functions working on entire arrays

- Mathematical and statistical functions functions applied to every element of an array
- Return scalars or arrays
- Axis may be specified
  - sum
  - mean
  - std, var
  - min, max
  - argmin, argmax
  - cumsum, cumprod cumulative sum (product) starting from 0 (1)
  - ...

Reading and writing arrays to/from files

- Text or binary format
- Text -> pandas
- Binary
    - raw, uncompressed, binary format
    - file name extension .npy
    - np.save(<path>, <array>)
    - np.load(<path>
    - It is possible to write/read many arrays from/to one file

Thank you for your attention!

- Let the power of Python be with you in 2021 (and the following years)!