Data Science: Numpy

CPSC 501: Advanced Programming Techniques
Fall 2022

Jonathan Hudson, Ph.D
Assistant Professor (Teaching)
Department of Computer Science
University of Calgary

Monday, October 3, 2022

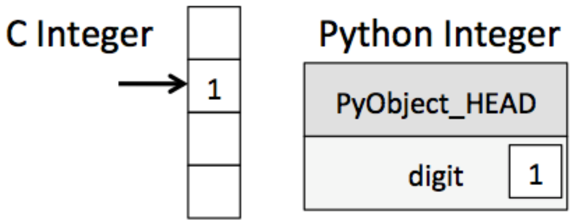
UNIVERSITY OF CALGARY

numpy

- **Num**erical **Py**thon library (2005 brought together disparate library ideas)
- More efficient data and storage operations as arrays grow larger
- **Python** integer is more than an integer, and a list more than just values

integers?

- **Python** integer is more than an integer
- **Python** 3.4 integer

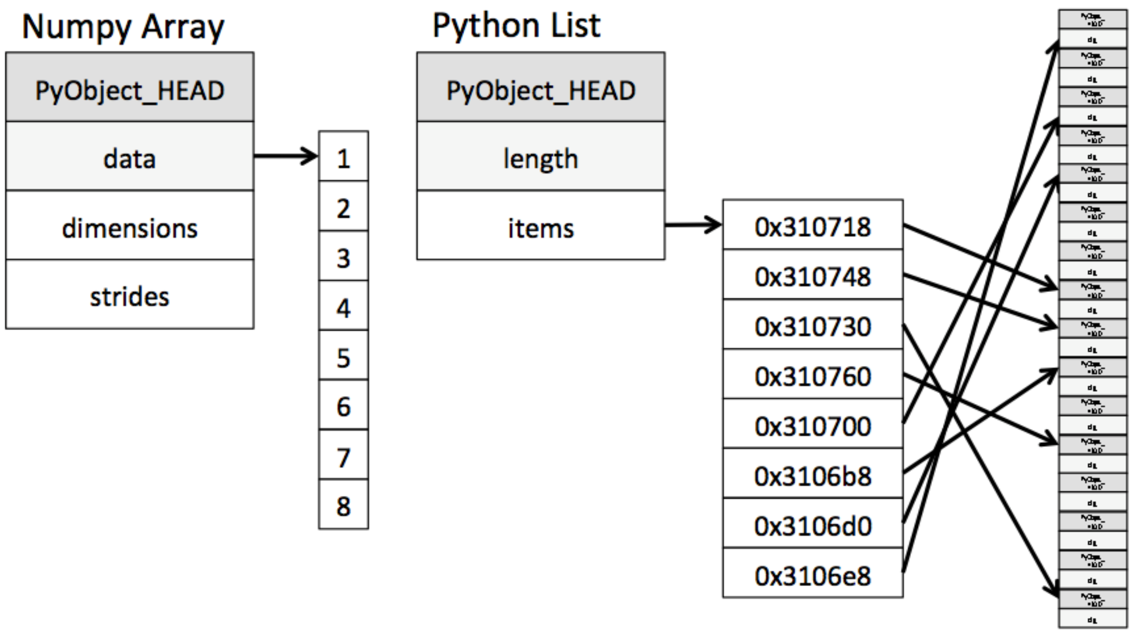



```c
struct _longobject {
  long ob_refcnt;
  PyTypeObject *ob_type;
  size_t ob_size;
  long ob_digit[1];
}
```

- A **C** integer is essentially a label for a position in memory whose bytes encode an integer value.
- A **Python** integer is a pointer to a position in memory containing all the Python object information, including the bytes that contain the integer value.

lists?

- A **Python** list is more than just a list of values



- A **Python** list contains a pointer to a block of pointers, each of which in turn points to a full **Python** object like the **Python** integer we saw earlier.
- **numpy** arrays are a single pointer to a block of contiguous data.

numpy

- **Num**erical **Py**thon library
- More efficient data and storage operations as arrays grow larger
- **Python** integer is more than an integer, and a list more than just values
- **numpy** arrays allow us to put data all in one place and drastically improve the our ability to manipulate it quickly.
- In essences what **numpy** does is provide a portal from **Python** through to **C** implementations of storage arrays, allowing us to access the strengths of that language in ways not normally available in **Python**.

```python
import numpy as np
```

array?

- **Python** does have its own **array** type

```python
import array
L = list(range(10))
A = array.array("i",L)
print(A)
```

```
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- There are similarities to representation that **numpy** also uses but **numpy** also adds a large range of efficient operations that the **array** type does not have
- Few people use **array** unless there is no way to import or access **numpy**

Create

- Numerous options to create **numpy** arrays

```
[ ] import numpy as np

[5] np.array(range(5))
array([0, 1, 2, 3, 4])

[12] np.array([1,2,3,4,5], dtype="float32")
array([1., 2., 3., 4., 5.], dtype=float32)

[11] np.zeros(10, dtype=int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

[13] np.full(3, 10)
array([10, 10, 10])

[16] np.random.random(3)
array([0.65720754, 0.97030252, 0.77293412])
```

Datatypes

- To gain the power of C arrays Python now has to care about all the types that a C programmer has to manage

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

Speed

- numpy operations speed comes from a large variety of Universal Functions (Ufuncs) which are ‘vectorized operations’ designed to run at higher speed than if we let Python’s loops manage things
- Both computer same answer!

```
[48] import numpy as np
     np.random.seed(0)

     def compute_reciprocals(values):
         output = np.empty(len(values))
         for i in range(len(values)):
             output[i] = 1.0 / values[i]
         return output

     values = np.random.randint(1, 10, size=5)

     print(compute_reciprocals(values))
     print(1.0 / values)
```

```
[0.16666667 1.         0.25       0.25       0.125     ]
[0.16666667 1.         0.25       0.25       0.125     ]
```

Speed

- Both computer same answer!
- Milliseconds versus microseconds (a difference in order of magnitude of 1000 here)

```
big_array = np.random.randint(1, 100, size=100000)
print("Python")
%timeit compute_reciprocals(big_array)
print("numpy")
%timeit (1.0 / big_array)
```

```
Python
1 loop, best of 5: 225 ms per loop
numpy
The slowest run took 4.95 times longer t
1000 loops, best of 5: 227 µs per loop
```

UFuncs

- Most **Ufuncs** are accessible using straight operators, although we can always use the longer form of np.divide for example
- Some don’t have operators like

np.abs(array)

np.sin(array), np.cos(array), etc.

np.log2(array), np.ln(array), etc.

np.sqrt(array), np.floor(array), etc.

- Others available via **scipy** under **special**

```
from scipy import special
```

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Boolean arrays

Operator	Equivalent ufunc	Operator	Equivalent ufunc
`==`	`np.equal`	`!=`	`np.not_equal`
`<`	`np.less`	`<=`	`np.less_equal`
`>`	`np.greater`	`>=`	`np.greater_equal`

Operator	Equivalent ufunc	Operator	Equivalent ufunc
`&`	`np.bitwise_and`	`|`	`np.bitwise_or`
`^`	`np.bitwise_xor`	`~`	`np.bitwise_not`

- Can use booleans to ask questions, combine booleans with bitwise operators
- Will product boolean arrays which can then be used as ‘masks’ for filtering

```
x = np.array([5, 2, 3, 4, 1])
print(x < 3)
print(np.any(x<4))
print(np.all(x<4))
print(np.sum(x<4))
print(np.sum((x < 4) & (x > 1)))
print(np.sum((x < 2) | (x > 3)))
print(x[(x < 2) | (x > 3)])
```

```
[False  True False False  True]
True
False
3
2
3
[5 4 1]
```

Save/Load

- **numpy** has both straight forward save/load, but also ability to save multiple in a form that multiple arrays can be reloaded and referenced by stored key

```
array = np.random.randint(0, 100, size = 15)
np.save('filename', array)
temp = np.load("filename.npy")
print(temp)
array2 = np.random.randint(0, 100, size = 15)
np.savez('filename2', a=array, b=array2)
temp1 = np.load("filename2.npz")
print(temp1['a'])
print(temp1['b'])
```

```
[59 83  3  6 33 96 72 54 36 29 81 85 95 51 65]
[59 83  3  6 33 96 72 54 36 29 81 85 95 51 65]
[27  3 62 15  0 47 56 96 79 40 88 14 92 96 89]
```

Onward to … pandas.

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY