

# Lecture 7: Data Representation, Numerical Python

October 16th, 2018

**Topics Introduced:** Binary vs. Decimal systems, simplified concepts of computer architecture including hierarchical memory and arithmetic logic units, representation of integers, strings, and floating point types, as well as `numpy`.

## 1 Representation of Data

### 1.1 Computer Representation of Integers

Computers represent and store everything in *binary*, which is a base 2 number system. How is this different from what we are familiar with in our base-10 (i.e. decimal) number system?

- Decimal counting uses the symbols 0-9.  
000, 001, 002, ..., 009
- When we exhaust the set of available symbols, we realize a type of overflow, in which the least-significant digit is reset to zero and the next digit of higher significance is incremented.  
010, 011, 012, ..., 099
- We repeat this process of allowing for overflow with each digit of significance.  
100, 101, 102, ...

To be formal, you can think of the integer  $d_md_{m-1} \dots d_1d_0$  as being equivalent shorthand for

$$d_0 \times 10^0 + d_1 \times 10^1 + \dots + d_{m-1} \times 10^{m-1} + d_m \times 10^m,$$

where each symbol  $d_i$  can take on a single value from the set  $\{0, 1, \dots, 9\}$ . *Counting in binary* is no different, sparing the fact that our set of symbols is reduced to have cardinality two, i.e. the atomic elements are *binary digits (bits)* which can be toggled between zero or one.

To be formal, we can think of the integer  $b_mb_{m-1} \dots b_1b_0$  as being equivalent shorthand for

$$b_0 \times 2^0 + b_1 \times 2^1 + \dots + b_{m-1} \times 2^{m-1} + b_m \times 2^m$$

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

Table 1: A table describing how values can be equivalently represented in either a decimal or a binary number system.

where here we are using  $b_i$ 's to remind ourselves that each symbol is a binary digit taking on only one of two values. Notice that other than the set of symbols changing, the *base* has changed as well, from 10 to 2. Once you learn these basics, [converting between numeral systems](#) is easy.

There are 8 bits in a [byte](#). Depending on the computer architecture, a [word](#) could be 4 or 8 bytes.

### 1.1.1 Simplified Model of Computer

Where are our data stored on a computer? How is it operated on? A typical computer will have a memory hierarchy to facilitate long-term storage (e.g. hard-disk), medium-term storage (RAM), and short-term storage (cache). We present a simplified computer architecture below, which does not picture long-term storage.

### 1.1.2 Common Prefixes

There can be some confusion here depending on the context of usage. That is, in the context of networking and storage, the prefixes **kilo**, **mega**, **giga**, **tera**, **peta**, and **exa** all use base 2. However, many other contexts speak in base 10, e.g. cpu-frequencies and ethernet speeds.

The prefix [kilo](#) always means one-thousand in the prefix of an [International System of Units](#). However, since we understand the difference between a **bit** and a **byte**, we can now understand that there is a difference between a **kilobyte** (denoted by **kB**) and a *kibibyte* (denoted by **KiB**). The former is 1,000 bytes, i.e. 8,000 bits, whereas the latter is 1,024 bytes. i.e. 8,192 bytes. We ask that you remember that there is a big difference between these terms, but not necessarily that you remember the exact terminology or symbols used to denote each. Wikipedia has a nice table demonstrating how binary and decimal systems use [different prefixes across orders of magnitude](#).

### 1.1.3 Computer Storage of an Integer

At the hardware level computers **don't** do variable length representations of numbers. I.e. although we may use a variable length representation, writing  $4_{10} = 100_2$ , or  $73_{10} = 1001001_2$ , computers often use a fixed width storage for numeric types.

At the hardware level computers typically handle integers using 8, 16, 32, or 64 bits, depending on the architecture.

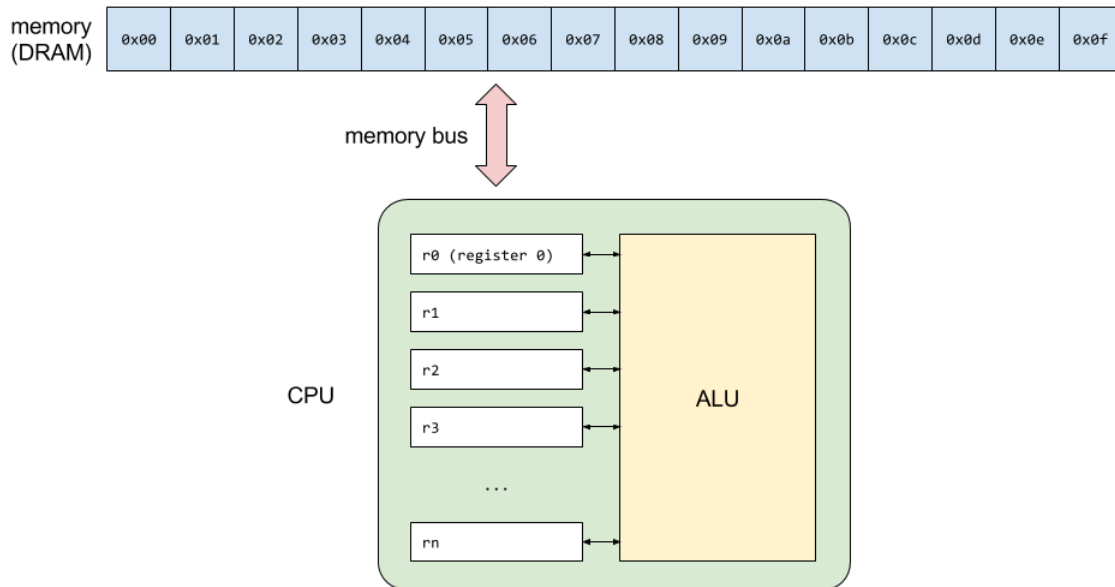


Figure 1: We take a simplified view of a computer architecture. In particular, we show the relationship between **primary memory** (e.g. **RAM**) and a **CPU**. The CPU contains (i) **arithmetic logic units** which take two inputs and produce a single output (think of the operators  $+$ ,  $-$ ,  $\&$ , and  $|$ ) and (ii) **registers** which can (very quickly) store inputs for ALU operations and hold corresponding outputs.

**Number of Representable Values Given  $n$  Bits** Realize that for a sequence of  $n$  bits, it may take on exactly  $2^n$  different values. This dictates the range of what can be represented with a fixed number of bits. Some common bounds we run into are.

$2^8$	= 256	
$2^{16}$	= 65536	~(65 thousand)
$2^{32}$	= 4294967296	~(4 billion)
$2^{64}$	= 18446744073709551616	~(18 followed by 18 zeros)

I.e. when we went from 32-bit machines to 64-bit machines, we didn't just double their capacity. The latter can store 10 orders of magnitude more information (in each register), akin to allowing 10 more decimal digits to be represented. This follows from the fact that

$$\log_{10}(2^{64}/2^{32}) = 64 \log_{10}(2) - 32 \log_{10}(2) = 32 \log_{10}(2) \approx 10.$$

**Using a Sign Bit to Store Negative Integers** The idea is to simply use one bit to store the sign of a value, and the remaining bits to track magnitude. This reduces the range

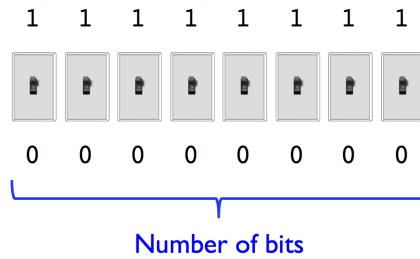


Figure 2: We can think of a (fixed-width) sequence of bits as a panel of lightswitches, each capable of being toggled on or off.

of the magnitude from  $2^n$  to  $2^{n-1}$ . E.g. suppose that we have a **nibble** to work with, i.e. four bits. We can represent the values  $\pm 4$  as indicated at the right of each statement.

$$\begin{aligned}
 4_{10} &= (100)_2 && \rightsquigarrow (0100) \\
 -4_{10} &= -1 \times (100)_2 && \rightsquigarrow (1100)
 \end{aligned}$$

**Offset: An Unconventional Way to Represent Negative Values** Another idea instead of using a sign bit is to simply apply an offset or bias to reinterpret the conversion between binary and decimal. One example may look as follows.

128	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1
-127	0	0	0	0	0	0	0

Figure 3: Suppose we have a byte or eight bits. There are  $2^8 = 256$  values that can be represented by such a sequence of bits. Centering the number of distinct values around the integer zero, we find that we can represent the set of values  $\{-127, \dots, 0, 1, \dots, 128\}$ . If we use an offset method for representing integers, then we may decide to set the sequence of eight bits as all zero's to represent the minimum value  $-127$ , and from here we can apply the usual rules of carry-over/overflow to count our way up to 128.

Again, whether we use the idiom of a sign-bit or if we apply an offset, either way we effectively reduce the range of the magnitude of integers allowed. Intuitively, whether or not a number is positive or negative should *require one unit of information*.

Many programming languages support unsigned integers. Although Python itself does not have **unsigned integers**, but **numpy** does. A programmer can use this to their advantage by expanding the effective range available if negative numbers don't need to be stored. But! With any fixed-storage representage of a real number, we must beware.

- Attempting to assign a value greater than what can be represented by the data type will result in *overflow*. Overflow tends to cause *wraparound*, e.g. if adding together two signed numbers causes overflow the result is likely to be a negative number.
- Attempting to assigning a value less than what can be represented by the data type will result in *underflow*.<sup>1</sup> Underflow can cause a division by zero error, for example, if we were to scale by the difference between two quantities  $\frac{1}{b-a}$  where  $a \neq b$  but  $a \approx b$ .

### 1.1.4 Range of Integer Types

For a fixed number of bits, if we choose to represent sequential integers centered around zero we must make a decision on boundaries. E.g. with eight bits, one can store  $2^8 = 256$  unique different bit-sequences (or values), where we have the choice of either being able to represent the values  $\{-128, \dots, 127\}$  or  $\{-127, \dots, 128\}$ . In general, an  $n$ -bit representation of an integer stores the values in the set

$$\underbrace{\{-2^{n-1}, \dots, -1\}}_{2^{n-1} \text{ elements}}, \underbrace{\{0, \dots, 2^{n-1} - 1\}}_{2^{n-1} \text{ elements}}.$$

**Integers in Python** In Python 3, values of type `int` may have unlimited range.<sup>2</sup> This is quite nice, and perhaps surprising if you come from other programming languages.

---

```

1 i = 52**100
2 print(type(i))      # <class 'int'>
3 print(i)             # This is beyond the 64-bit integer range.

```

---

The module `numpy` supports *only* fixed-width integers for reasons that relate to both performance and storage.

## 1.2 Strings

### 1.2.1 ASCII

American Standard Code for Information Interchange (ASCII) is typically used to encode text information. Characters, numbers, symbols, etc. are encoded using 7 bits (although on modern computers they would typically use 8 bits).

E.g. A maps to  $(1000001)_2 = (65)_{10}$ , and B maps to  $(1000010)_2 = (66)_{10}$ .

Default Python 2 string is ASCII. Possible to get *Unicode* strings with `s = u'I am a unicode string!'`.

---

<sup>1</sup>We can try to fill the underflow gap using *denormal numbers*.

<sup>2</sup>Python 2 had both fixed size integers (with type `int`) and variable width integers (with type `long`).

### 1.2.2 Why Not ASCII All The Way Down? Motivating UTF-8

**For English, A Single Byte Suffices** The English language has 26 letters (52 if we include casing). If we throw in punctuation and digits 0-9, we realize that we have more than 64 unique characters to represent. I.e. we need at least 7 bits. Since computers are more efficient at processing bit sequences aligned a particular way, many implementations use 8-bits for an ASCII set.

**For Languages with Ideographs, Require 2 Bytes** But what about other languages? Chinese, Japanese, and Korean have somewhere between  $2^8$  and  $2^{16} - 1$  unique characters, whence we require a two-byte sequence to store each character in these languages. So, different languages (or regions) require different *character encodings* to map symbols (text) to bit-sequences.

**For World of All Languages, (naively) Require 4 Bytes** But what about the world wide web? The number of unique characters across all languages exceeds 65,535, whence we require strictly more than two-bytes to encode any character coming from any language in the world. Sticking with powers of two, we'll use four-bytes. But all of a sudden, we realize how expensive this has become; any given language only requires one or at most two bytes per character, but we're using double that.

The advantage to UTF-32 (the 4-byte unicode encoding) despite it being costly in terms of storage is that because each character is represented as a fixed width sequence of bits, we can slice into strings in constant time. I.e. looking up the  $n$ th character in a string is easy, we simply look up where the object lives in memory and jump  $n$  characters forward in memory, where each character corresponds to exactly 32-bits.

**Using a Variable Length Encoding  $\rightsquigarrow$  Efficiency** Alas, we've followed Mark Pilgrim's motivation of UTF-8 [as per Chapter 4 of "Dive Into Python"](#). UTF-8 is a *variable length encoding*. The implication is that the set of ASCII characters each requires only a single byte per character. Latin characters such as ä and ñ may require two bytes per character, Chinese characters may involve three bytes, and rarely used characters require four bytes. However, since each character requires a different number of bits, we can *no longer* jump to the  $k$ th character of a string with  $O(1)$  work; we're relegated to inspecting the first  $k - 1$  characters to find out where in the bit-sequence the  $k$ th character is represented.

**Default String Encoding in Python is UTF-8** Default string encoding in Python 3 is [UTF-8](#). The change from ASCII to Unicode between Python 2 and 3 caused major headaches for Python community. The fact that each character of text may take from 1 to 4 bytes is nice in that the 1-byte codes correspond to ASCII characters, which makes UTF-8 backwards compatible with ASCII. Currently, UTF-8 encodes a total of 1,112,064 characters, which is enough to represent the majority of human character systems.

## 1.3 Floating Point Representation of Numeric Values

How do we represent a floating point value using bits?

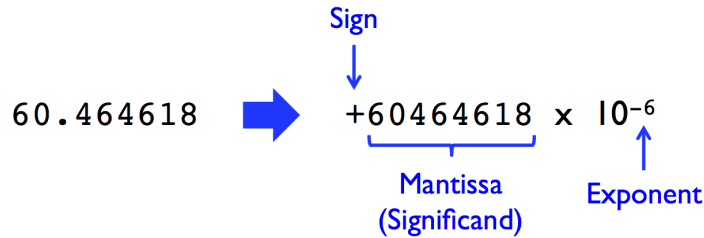


Figure 4: A representation of a decimal value that is broken down into three pieces: (i) a sign bit, (ii) a fractional component, and (iii) an exponent indicating the magnitude of rescaling. This is a generic recipe for approximating real numbers using a finite set of bits.

### 1.3.1 Floating Point Standard

IEEE (Institute of Electrical and Electronics Engineers) 754 is the technical standard for floating point used by all modern processors. The standard also specifies things like rounding modes, handling overflow, divide by zero, etc.

	Sign bits	Exponent bits	Mantissa bits
32 bit floating point	1	8	23
64 bit floating point	1	11	52

Figure 5: 32 vs. 64-bit floating point formats. In moving from 32-bit to 64-bit standards, IEEE increased the number of bits allocated to the mantissa significantly more than they did the exponent: with 11-bits for the exponent, we can represent  $2^{11} = 2,048$  different values of exponents. We center these exponents around zero using a [exponent bias](#), such that we can store exponents in the range  $\pm 2^{10}$ . But these are exponents, i.e. they are used to raise our base to a specified power. That's an astronomically large rescaling.

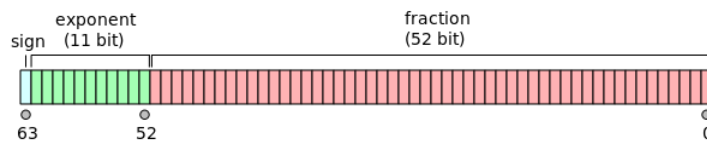


Figure 6: A visual layout of the 64 bit representation used to store a floating point value. We have 1 sign bit, 52 mantissa (significand, fractional) bits, and 11 exponent bits. Source: [Wikipedia](#).

Symbolically,

$$(-1)^{\text{sgn}} \underbrace{\left( 1 + \sum_{i=1}^{52} b_{52-i} 2^{-i} \right)}_{=(1.b_{51}b_{50}\dots b_0)_2} \times 2^{\text{exp}-1023}$$

where the exponent term `exp` is simply an integer value represented using 11 bits in a vanilla fashion, and we're subtracting 1,023 as means of **exponent bias** such that we can have both positive and negative exponents without requiring the use of an *additional* sign-bit for just the exponent term.

**Floating Point and You** Floating point also has similar potential for overflow and underflow. In addition, the limited number of bits for the mantissa means it often needs to be rounded. We'll spend more time on floating point arithmetic in CME 212. Canonical resource: "What Every Computer Scientist Should Know About Floating-Point Arithmetic" by Goldberg ([link](#)). Floating point numbers in Python are double precision (64-bit). Additionally, `numpy` has support for 16-bit and 32-bit floating point formats.

### 1.3.2 Correctness

#### Some disasters attributable to bad numerical computing

By Douglas N. Arnold. <http://www.math.umn.edu/~arnold/disasters/disasters.html>

Have you been paying attention in your numerical analysis or scientific computation courses? If not, it could be a costly mistake. Here are some real life examples of what can happen when numerical algorithms are not correctly applied.

The [Patriot Missile failure](#), in Dhahran, Saudi Arabia, on February 25, 1991 which resulted in 28 deaths, is ultimately attributable to poor handling of rounding errors.

The [explosion of the Ariane 5 rocket](#) just after lift-off on its maiden voyage off French Guiana, on June 4, 1996, was ultimately the consequence of a simple overflow.

The [sinking of the Sleipner A offshore platform](#) in Gandsfjorden near Stavanger, Norway, on August 23, 1991, resulted in a loss of nearly one billion dollars. It was found to be the result of inaccurate finite element analysis.

See also: <https://people.eecs.berkeley.edu/~wkahan/> for thoughts from the **father of floating point**, i.e. the primary architect of an IEEE standard for representing floating point values in a computer architecture.

## 2 Python for Numerical Computing

### 2.1 Motivation: Python is (Relatively) Slow

One of the main disadvantages of a higher level language is that, while comparatively easy to program, it offers less direct control over resources; an experienced programmer using C++ or other lower level languages may be able to write a more performant program.



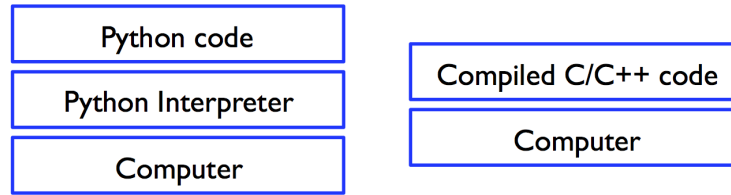


Figure 7: The Python interpreter sits between the programmer’s code and the machine. In the case of a lower level language such as C++, the programmer has direct control over memory management and more granular control over data types. A knowledgeable programmer can take advantage of the particular computer architecture they are using, alongside the problem constraints, in order to more efficiently write code to solve a task. This is harder with Python, wherein the code that the programmer writes is first translated by the interpreter such that the machine may understand it. We remark that of course C++ code also must be translated to machine code, but that it ties in much closer to said assembly code than Python.

### 2.1.1 Runtime Performance: An Empirical Example

Let’s compute  $2^i$  for  $i \in [0, n)$ . We can use the `timeit` module. Using lists, and from the command line interface.

```
$ python3 -m timeit --setup='L=range(1000)' '[i**2 for i in L]'
```

Equivalently, we can use the `%timeit` magic command in an IPython notebook.

```
$ ipython
In [1]: L = range(1000)
In [2]: %timeit [i**2 for i in L]
```

What happens if we try the same operation using `numpy`? The analogous operation to `range` in built-in Python is the `arange` method in `numpy`. For integer arguments, the two functions behave the same, but `numpy.arange` extends to handling non-integer arguments.<sup>3</sup>

---

```
1 import numpy as np
2 a = np.arange(1000)
3 %timeit a**2
```

---

On my machine, the difference is about 300x. We’ll explore how `numpy` is able to achieve such performance gains throughout the lecture. For now, let’s continue to explore how the module can not only allow our programs to execute faster once built, but in fact `numpy` lets us write the more concise code that is capable of a higher level of abstraction.

### 2.1.2 Programmer Productivity

And even though Python is a high-level programming language, it’s still a general purpose one. I.e. it is not tailored to numerical or scientific computing. Whence we *do* have to write

---

<sup>3</sup>The `range` sub-routine is designed to return *indices* that can be used as subscript arguments when slicing an object, whereas the `numpy.arange` is a more generalized function which can take arbitrary boundary points and step-sizes.

additional code in order to abstract away commonly used operations in maths and sciences.

E.g. let's add some 2D arrays. In Python, we could try and create a two-dimensional tabular data structure using lists of lists. In order to facilitate something as easy as “adding” two 2D-arrays together, we'd have to write our own sub-routine, since the `+` operator performs *concatenation* on lists rather than element wise addition!

---

```

1  def my_ones(nrows, ncols):
2      """
3      Create a matrix-like object of dimension nrows x ncols, with each
4      element taking unit value.
5      """
6      A = []
7      for r in range(nrows):
8          A.append([])
9          for c in range(ncols):
10             A[r].append(1.0)
11     return A
12
13 def matrix_add(A,B):
14     """
15     Create a new matrix-like object to store the result of adding
16     input matrices A and B. We assume the matrices have identical
17     dimension. The work required by the algorithm is proportional to
18     the number of elements stored in the input matrices.
19     """
20     C = []
21     for r in range(len(A)):
22         C.append([])
23         for c in range(len(A[r])):
24             C[r].append(A[r][c] + B[r][c])
25     return C
26
27 nrows, ncols = 3, 2
28 A = my_ones(nrows,ncols)
29 B = my_ones(nrows,ncols)
30 C = matrix_add(A,B)

```

---

What about with numpy? The operations are trivial to carry out without defining our own data structures or sub-routines. I.e. we can “program faster”.

---

```

1  nrows = 3
2  ncols = 2
3
4  A = np.ones((nrows,ncols))
5  B = np.ones((nrows,ncols))
6
7  C = A + B

```

---

Seems a lot easier to write. Let's check if we earned ourselves a performance again. Suppose we have placed our definition of `my_ones` into a standalone file in our current working directory,

which we call `my_module.py`. We may time the performance as follows, using

```
$ python3 -m timeit --setup='import my_module' 'my_module.my_ones(1000, 500)'
```

Alternatively, in an IPython notebook we may simply write `%timeit A = my_ones(1000,500)`. `numpy` is able to accomplish the analogous operation on my machine approximately 200x faster. I.e. whereas the hand-rolled variant required 50 *milliseconds* per loop, the `numpy` analogue below requires only 200 *microseconds* per loop.<sup>4</sup>

```
$ python3 -m timeit --setup='import numpy as np' 'np.ones((1000, 500))'
```

What about our add operation? In a IPython notebook, with `my_ones` as per above.

---

```
1 nrows = 1000
2 ncols = 500
3
4 A = my_ones(nrows,ncols)
5 B = my_ones(nrows,ncols)
6 %timeit C = matrix_add(A,B)
7
8 A = np.ones((nrows,ncols))
9 B = np.ones((nrows,ncols))
10 %timeit C = A + B
```

---

Again we see that `numpy` is orders of magnitude faster.

**Object Overhead** The reason that

```
>>> a = [[0,1,0],[1,0,1],[0,1,0]]
>>>
```

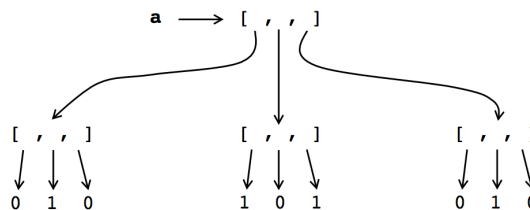


Figure 8: fig

## 2.2 Options for better performance

Python is great for quick projects, prototyping new ideas, etc. but what if you need better performance? One option is to completely rewrite your program in something like C or C++

<sup>4</sup>There are 1,000 microseconds in a millisecond.

### 2.2.1 Python C API

Python has a C API which allows the use of compiled modules

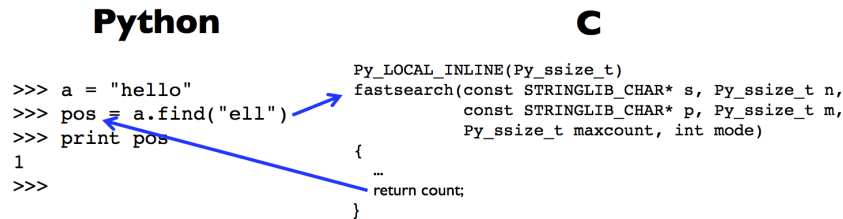


Figure 9: fig

The actual implementation of `string.find()` can be viewed at:

[stringlib/fastsearch.h](https://stringlib/fastsearch.h).

### 2.2.2 Python compiled modules

Python code in a `.py` file is actually executed in a hybrid approach by a mix of the interpreter and compiled modules that come with Python

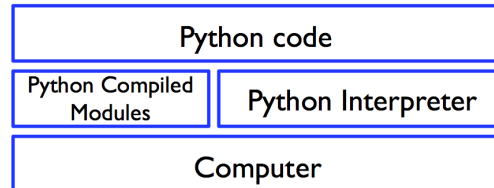


Figure 10: fig

### 2.2.3 Extension modules

- The same Python C API used by the developers of Python itself also allows other programmers to develop and build their own compiled extension modules
- These modules extend the functionality of Python with high performance implementations of common operations
- Other languages, such as C++ and Fortran, are also supported by using the C API

### 2.2.4 NumPy, SciPy, matplotlib

- NumPy - multidimensional arrays and fundamental operations on them

- SciPy - Various math functionality (linear solvers, FFT, optimization, etc.) utilizing NumPy arrays
- matplotlib - plotting and data visualization
- None of these packages seek to clone MATLAB, if you want that try something like GNU Octave

### 2.2.5 Python software stack

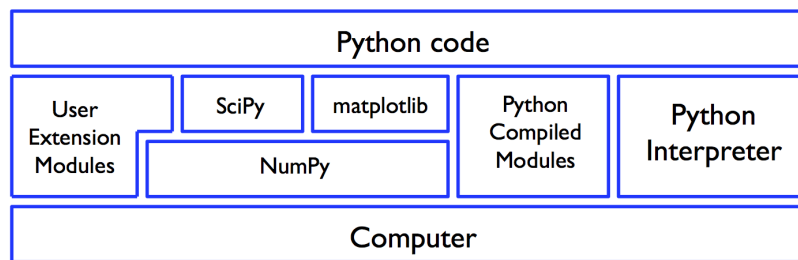


Figure 11: fig

## 2.3 Motivation

## 2.4 NumPy

NumPy provides a numeric array object:

```

1 \begin{Highlighting}[]
2 \ImportTok{import}\NormalTok{ numpy }\ImportTok{as}\NormalTok{ np}
3 \NormalTok{a }\OperatorTok{=}\NormalTok{ np.array([ }\DecValTok{7}\NormalTok{ }, }\DecValTok{
4 \BuiltInTok{print}\NormalTok{(a)}
5 \end{Highlighting}
  
```

```

1 \begin{Highlighting}[]
2 \NormalTok{a[ }\DecValTok{1}\NormalTok{ ]}
3 \end{Highlighting}
  
```

```

1 \begin{Highlighting}[]
2 \NormalTok{a[ }\DecValTok{1}\NormalTok{ ] }\OperatorTok{=} \DecValTok{19}
3 \NormalTok{a}
4 \end{Highlighting}
  
```

### 2.4.1 Arrays are not lists

---

```

1 \begin{Highlighting}[ ]
2 \NormalTok{a}[ ]\DecValTok{0}\NormalTok{[ ] }\OperatorTok{=} \StringTok{"hello"}
3 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[ ]
2 \NormalTok{a.append( )}\DecValTok{8}\NormalTok{( )}
3 \end{Highlighting}

```

---

### 2.4.2 NumPy arrays

- NumPy arrays contain **homogeneous** data (all elements must have same type)
- Size is fixed, i.e. you can't append or remove

### 2.4.3 Data types

- Integers
  - 8, 16, 32, and 64 bit signed and unsigned (np.int8, np.uint8, etc.)
- Floating point
  - 32, 64, 128 bit (np.float32, np.float64, etc.)
- Complex, strings, and Python object references also supported

### 2.4.4 Data type examples

---

```

1 \begin{Highlighting}[ ]
2 \NormalTok{a }\OperatorTok{=} \NormalTok{ np.array( [ ]\DecValTok{7}\NormalTok{[ , ]\DecValTok{0}\NormalTok{ }
3 \NormalTok{a}
4 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[ ]
2 \NormalTok{a}[ ]\DecValTok{0}\NormalTok{[ ] }\OperatorTok{=} \NormalTok{ a}[ ]\DecValTok{0}\NormalTok{ }
3 \NormalTok{a}
4 \end{Highlighting}

```

---

---

```

1 \begin{Highlighting}[
2 \NormalTok{b }\OperatorTok{=}\NormalTok{ np.array([}\DecValTok{4}\NormalTok{, }\DecValTok{
3 \NormalTok{b}
4 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[
2 \NormalTok{b[}\DecValTok{0}\NormalTok{] }\OperatorTok{=} \DecValTok{437}
3 \NormalTok{b}
4 \end{Highlighting}

```

---

### 2.4.5 Multidimensional arrays

- Arrays can have multiple dimensions called *axes*
- The number of *axes* is called the *rank*
- These terms come from the NumPy community and should not be confused with linear algebra terms for *rank*, etc.

### 2.4.6 Multidimensional arrays

---

```

1 \begin{Highlighting}[
2 \NormalTok{a }\OperatorTok{=} \NormalTok{ np.array([}\DecValTok{7}\NormalTok{, }\DecValTok{
3 \NormalTok{a}
4 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[
2 \NormalTok{a.ndim}
3 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[
2 \NormalTok{a.dtype}
3 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[
2 \NormalTok{a.shape}
3 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[
2 \NormalTok{a.size}
3 \end{Highlighting}

```

---

### 2.4.7 Internal representation

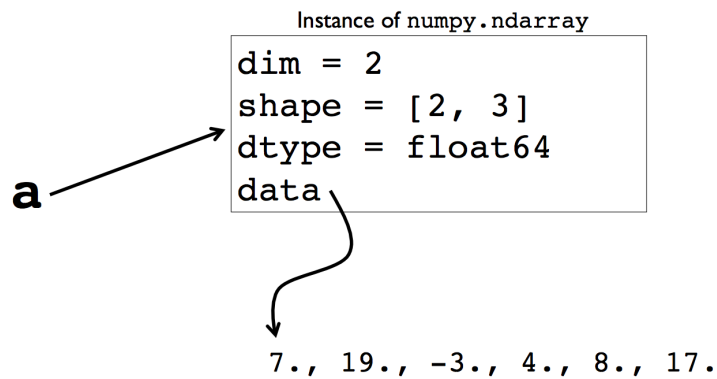


Figure 12: fig

### 2.4.8 Creating arrays

---

```

1 \begin{Highlighting}[
2 \NormalTok{a } \OperatorTok{=} \NormalTok{ np.empty( ( \DecValTok{3} \NormalTok{ {,} \DecValTok{
3 \NormalTok{a}
4 \end{Highlighting}
  
```

---



---

```

1 \begin{Highlighting}[
2 \NormalTok{a } \OperatorTok{=} \NormalTok{ np.zeros( ( \DecValTok{3} \NormalTok{ {,} \DecValTok{
3 \NormalTok{a}
4 \end{Highlighting}
  
```

---



---

```

1 \begin{Highlighting}[
2 \NormalTok{a } \OperatorTok{=} \NormalTok{ np.ones( ( \DecValTok{3} \NormalTok{ {,} \DecValTok{
3 \NormalTok{a}
4 \end{Highlighting}
  
```

---



---

```

1 \begin{Highlighting}[
2 \NormalTok{a } \OperatorTok{=} \NormalTok{ np.eye( \DecValTok{3} \NormalTok{ {) }
3 \NormalTok{a}
4 \end{Highlighting}
  
```

---



---

```

1 \begin{Highlighting}[
2 \NormalTok{a } \OperatorTok{=} \NormalTok{ np.arange( \DecValTok{9} \NormalTok{ {, dtype} \OperatorTok{=} \NormalTok{
3 \NormalTok{a}
4 \end{Highlighting}
  
```

---





---

```

1 \begin{Highlighting}[]
2 \NormalTok{a }\OperatorTok{=}\NormalTok{ np.arange(}\DecValTok{9}\NormalTok{, dtype}\OperatorTok{ }\DecValTok{0}
3 \NormalTok{a}
4 \end{Highlighting}

```

---

```

1 \begin{Highlighting}[]
2 \CommentTok{# a slice}
3 \NormalTok{a[}\DecValTok{3}\NormalTok{:}\DecValTok{7}\NormalTok{]}
4 \end{Highlighting}

```

---

```

1 \begin{Highlighting}[]
2 \CommentTok{# assign to a slice}
3 \NormalTok{a[}\DecValTok{3}\NormalTok{:}\DecValTok{7}\NormalTok{]} \OperatorTok{=}\DecValTok{0}
4 \NormalTok{a}
5 \end{Highlighting}

```

---

```

1 \begin{Highlighting}[]
2 \DecValTok{2}\OperatorTok{*}\NormalTok{a}
3 \end{Highlighting}

```

---

```

1 \begin{Highlighting}[]
2 \NormalTok{a}\OperatorTok{*}\NormalTok{a}
3 \end{Highlighting}

```

---

```

1 \begin{Highlighting}[]
2 \BuiltInTok{sum}\NormalTok{(a)}
3 \end{Highlighting}

```

---

```

1 \begin{Highlighting}[]
2 \BuiltInTok{min}\NormalTok{(a)}
3 \end{Highlighting}

```

---

```

1 \begin{Highlighting}[]
2 \BuiltInTok{max}\NormalTok{(a)}
3 \end{Highlighting}

```

---

## 2.4.12 Array operations

---

```

1 \begin{Highlighting}[]
2 \NormalTok{a }\OperatorTok{=}\NormalTok{ np.arange(}\DecValTok{9}\NormalTok{, dtype}\OperatorTok{ }\DecValTok{0}
3 \BuiltInTok{print}\NormalTok{(a)}
4
5 \CommentTok{# bad idea}

```

---

```

6 \NormalTok{total }\OperatorTok{=} \FloatTok{0.}
7 \ControlFlowTok{for}\NormalTok{ n }\KeywordTok{in} \BuiltInTok{range}\NormalTok{({}\BuiltInTok{len}\NormalTok{ a})}
8 \NormalTok{    total }\OperatorTok{+=}\NormalTok{ a[n]}\OperatorTok{*}\NormalTok{ a[n]}
9
10 \NormalTok{math.sqrt(total)}
11 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[]
2 \CommentTok{# better idea}
3 \ImportTok{import}\NormalTok{ math}
4 \NormalTok{math.sqrt(np.dot(a,a))}
5 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[]
2 \CommentTok{# best idea}
3 \NormalTok{np.linalg.norm(a)}
4 \end{Highlighting}

```

---

### 2.4.13 Speed of array operations

---

```

1 \begin{Highlighting}[]
2 \OperatorTok{%}\NormalTok{timeit total }\OperatorTok{=} \BuiltInTok{sum}\NormalTok{(np.arange(1000000))}
3 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[]
2 \OperatorTok{%}\NormalTok{timeit total }\OperatorTok{=} \NormalTok{ np.sum(np.arange(1000000))}
3 \end{Highlighting}

```

---

### 2.4.14 Loops vs. array operations

- Loops you write in Python will be executed by the interpreter
- Some of the overloaded operators (e.g. `min`, `max`, `sum`, etc.) work albeit slowly
- Calling NumPy function or methods of the array object will invoke high performance implementations of these operations

### 2.4.15 Matrix operations

---

```

1 \begin{Highlighting}[]
2 \NormalTok{a }\OperatorTok{=} \NormalTok{ np.arange(10, dtype=np.float64)}
3 \NormalTok{a}
4 \end{Highlighting}

```

---

---

```

1 \begin{Highlighting}[]
2 \NormalTok{a.transpose()}
3 \end{Highlighting}

```

---

```

1 \begin{Highlighting}[]
2 \NormalTok{np.trace(a)}
3 \end{Highlighting}

```

---

```

1 \begin{Highlighting}[]
2 \NormalTok{a}\OperatorTok{*}\NormalTok{a }\CommentTok{# element wise multiplication}
3 \end{Highlighting}

```

---

```

1 \begin{Highlighting}[]
2 \NormalTok{np.dot(a,a) }\CommentTok{# matrix-matrix multiplication}
3 \end{Highlighting}

```

---

```

1 \begin{Highlighting}[]
2 \CommentTok{# new matrix multiply operator in Python 3.5}
3 \NormalTok{a }\OperatorTok{@}\NormalTok{a}
4 \end{Highlighting}

```

---

#### 2.4.16 array vs matrix

- NumPy has a dedicated matrix class
- However, the matrix class is not as widely used and there are subtle differences between a 2D array and a matrix
- It is highly recommended that you use 2D arrays for maximum compatibility with other NumPy functions, SciPy, matplotlib, etc.
- See here for more details:

[http://www.scipy.org/NumPy\\_for\\_Matlab\\_Users](http://www.scipy.org/NumPy_for_Matlab_Users)

(array' or matrix'? Which should I use?)

#### 2.4.17 References to an array

---

```

1 \begin{Highlighting}[]
2 \NormalTok{a }\OperatorTok{=}\NormalTok{ np.arange()}\DecValTok{9}\NormalTok{, dtype}\OperatorTok{=}
3 \NormalTok{a}
4 \end{Highlighting}

```

---

---

```

1 \begin{Highlighting}[]
2 \NormalTok{b }\OperatorTok{=}\NormalTok{ a}
3 \NormalTok{b[]}\DecValTok{0}\NormalTok{,}\DecValTok{0}\NormalTok{[] }\OperatorTok{=}\DecValTok{0}
4 \NormalTok{b}
5 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[]
2 \NormalTok{a}
3 \end{Highlighting}

```

---

### 2.4.18 Array slices and references

---

```

1 \begin{Highlighting}[]
2 \NormalTok{a }\OperatorTok{=}\NormalTok{ np.arange()}\DecValTok{9}\NormalTok{, dtype}\OperatorTok{=}
3 \NormalTok{a}
4 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[]
2 \NormalTok{b }\OperatorTok{=}\NormalTok{ a[]}\DecValTok{2}\NormalTok{:}\DecValTok{7}\NormalTok{, dtype}\OperatorTok{=}
3 \NormalTok{b}
4 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[]
2 \NormalTok{b[]}\DecValTok{2}\NormalTok{[] }\OperatorTok{=}\DecValTok{-1}
3 \NormalTok{b}
4 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[]
2 \NormalTok{a}
3 \end{Highlighting}

```

---

### 2.4.19 Array copies

---

```

1 \begin{Highlighting}[]
2 \NormalTok{a }\OperatorTok{=}\NormalTok{ np.arange()}\DecValTok{9}\NormalTok{, dtype}\OperatorTok{=}
3 \NormalTok{a}
4 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[]
2 \NormalTok{b }\OperatorTok{=}\NormalTok{ a.copy()}
3 \NormalTok{b}
4 \end{Highlighting}

```

---

---

```

1 \begin{Highlighting}[
2 \NormalTok{b}[ ]\DecValTok{4}\NormalTok{[ ] }\OperatorTok{=} \DecValTok{-1}
3 \NormalTok{b}
4 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[
2 \NormalTok{a}
3 \end{Highlighting}

```

---

#### 2.4.20 Universal functions (ufuncs)

---

```

1 \begin{Highlighting}[
2 \ImportTok{import}\NormalTok{ numpy}
3 \NormalTok{a }\OperatorTok{=}\NormalTok{ np.arange(}\DecValTok{9}\NormalTok{[, dtype}\OperatorTok{=}\NormalTok{ np.float64)}
4 \NormalTok{a}
5 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[
2 \ImportTok{import}\NormalTok{ math}
3 \NormalTok{math.sqrt(a)}
4 \end{Highlighting}

```

---



---

```

1 \begin{Highlighting}[
2 \NormalTok{np.sqrt(a)}
3 \end{Highlighting}

```

---

#### 2.4.21 Beyond just arrays

- NumPy has some support for some useful operations beyond the usual vector and matrix operations:
  - Searching, sorting, and counting within arrays
  - FFT (Fast Fourier Transform)
  - Linear Algebra
  - Statistics
  - Polynomials
  - Random number generation
- SciPy has largely replaced much of this functionality, plus added much more

### 2.4.22 Warning

- Once you start making use of extension modules such as NumPy, SciPy, etc. the chances of code “breaking” when you run it on different machines goes up significantly
- If you do some of your development on machines other than corn (which isn’t the model we advise) you may run into issues

### 2.4.23 Further Reading

- MATLAB users: [http://www.scipy.org/NumPy\\_for\\_Matlab\\_Users](http://www.scipy.org/NumPy_for_Matlab_Users)
- NumPy tutorial at: [http://www.scipy.org/Tentative\\_NumPy\\_Tutorial](http://www.scipy.org/Tentative_NumPy_Tutorial)
- Official docs at: <http://docs.scipy.org/>