QUICK PYTHON TUTORIAL NUMBER TWO: DATATYPES AND ORGANIZATIONAL STRUCTURES

January 21, 2018

Contents

By the term *datatype* we mean, for example, integers, floating point variables, and complex numbers. By the term *organizational structures* we mean tuples, arrays and dictionaries. We cover these in the following sections. All available datatypes can be arranged in all available organizational structures. For example, we can have arrays of integers, dictionaries of complex numbers.

1. DATATYPES

Here we cover only the basic Python datatypes. There are others, including unsigned integers and complex numbers.

1.1. DIGITS, BITS, BYTES, AND WORDS

We have gotten to the place where you need to know a little about the internal workings of computers. Specifically, how the computer stores numbers and characters.

Humans think of numbers expressed in powers-of-ten, or *decimal numbers*. This means that there are 10 digits ($0 \rightarrow 9$) and you begin counting with these digits. When you reach the highest number expressible by a single digit, you use two digits and generate the next series of numbers, $10 \rightarrow 99$. Let f and s be the *first* (1) and *second* (0) digits, respectively; then the number is $f * 10^1 + s * 10^0$. And so on with more digits.

Fundamentally, all computer information is stored in the form of *binary numbers*, meaning powers-of-two. How many digits? Two! They are 0 and 1. The highest number expressible by a single digit is 1. The two-digit numbers range from 10 to 11; the number is $f * 2^1 + s * 2^0$. And so on with more digits. But wait a minute! The word "digit" is a misnomer—it implies something about 10 fingers. Here it's the word **bit** that counts. Each binary "digit" is really a **bit**. So the binary number 1001 is a 4-bit number. What decimal number does the binary number 1001 equal?

For convenience, computers and their programmers group the bits into groups of eight. Each group of 8 bits is called a **byte**. Consider, then, the binary number 11111111; it's the maximum-sized number that can be stored in a byte. What is this number?

Finally, computers group the bytes into **words**. The oldest PC's dealt with 8-bit words—one byte. Many present-day PCs use 32-bit words—four bytes. Macs and the more capable PCs deal with 64-bit words—8 bytes. What's the largest number you can store in a 4-byte word? And how about negative numbers?

Below we describe how Python (and everybody else) gets around this apparent upper limit on numbers. They do this by defining different data types. Up to now, the details didn't matter much. But now... We don't cover all datatypes below—specifically, we omit a detailed discussion on the built-in datatypes in Python, but you can look these up if you are interested.

2. NUMERIC DATATYPES IN PYTHON AND NUMPY

2.1. Native Numbers in Python

The datatypes *Python 2.7* natively supports are limited to int, long, float and complex. The datatypes are automatically fixed during variable assignment and need not be specified during declaration. However, remember that they do not change! The numeric datatypes are immutable, meaning that $1/2 = 0$ and not 0.5.

Integer numbers that can be represented by up to 32 bits are represented by the *int* datatype. They can be both signed and unsigned. Integer numbers that are larger, are represented by *long.* Unlike lower level languages (IDL, C), Python does not have a cap on the size of the number. It is capable of allocating as many bits are required to represent the number. *Float* is used to represent fractions up to 64 bits. We will see how fractions are stored later…

For most scientific computations you will proably use NumPy which has a lot more flexibility on the number of bits allocated to a number. This is more useful for programmers who deal with large sets of data (like radio astronomers!), and want to control the amount of memory used and optimise performance.

2.2. Integer Datatypes in NumPy

NumPy supports a much wider range of numeric datatypes than native Python. The length of the word and the precision of the fractional bit determine the memory required for representing a number. The shorter the word, the less memory required; the longer the word, the larger the numbers can be. Different requirements require different compromises.

2.2.1. 1 byte: The Boolean Datatype, `uint8` and `int8`

The `bool` datatype is a single byte long and can be `True` or `False`. Empty string " " and 0 are interpreted as `False` and everything else is is considered `True`. This datatype is most useful for evaluating conditional statements.

`uint8` stands for unsigned integer that can be represented in 8 bits.Therefore, its value runs from $0 \rightarrow 255$. `int8` stands for signed integer. The first bit is reserved to store the sign of the number and the rest 7 bits are used to represent the integer. Therefore, its value runs from $-127 \rightarrow 127$.

The byte data is not used as frequently anymore since computers have become faster. In the past, computers used to store colors of pixels in bytes (1 bytes each for red, blue and green) and encode characters (like the string variable 'a') in bytes. Python being a higher level language than

Fortran, IDL or C stores both colors and characters in custom formats. These are still numbers under the hood, but the user never sees these representations. The place where byte data is still frequently used is in analog to digital convertors; most ADCs quantise the analog signal into 8 bits. The one we have in the lab is more sophisticated and uses 16 bits.

If you ever need to generate a byte array, you can use

```
np.linspace(startvalue,stopvalue,dtype=np.uint8)
```

Remember that if, during a calculation, a byte number exceeds 255, then it will "wrap around"; for example, 256 wraps to 0, 257 to 1, etc.

2.2.2. Multiple bytes

Most frequently for computation, you need numbers greater than 255. NumPy offers integer representations with 2,4 and 8 bytes (16 bit, 32 bit and 64 bit numbers). In each representation you can choose to have just positive numbers, called **Unsigned Integers**. Unsigned numbers range from $0 \rightarrow 2^B - 1$, where B is the number of bits. How do you think unsigned integers wrap around?

Normally, you want the possibility of negative numbers and you can use **Integers**. If B is the number of bits, the total number of integer values is 2^B. One possible value is, of course, zero. So the number of negative and positive values differ by one. The choice is to favor negative numbers, so Integers cover the range $-2^{B/2} \rightarrow 2^{B/2} - 1$.

In general, you can generate arrays of various sizes (8,16,32,64 bits) using `dtype=np.uint<bits>` for unsigned integers and `dtype=np.int<bits>` for signed integers. What are the limits on these numbers? See Python help under "NumPy DataTypes" for more information.

2.3. FLOATING DATATYPES IN IDL

The problem with integer datatypes is that you can't represent anything other than integral numbers—no fractions! Moreover, if you divide two integer numbers and the result be fractional, but it won't be; instead, it will be rounded down (e.g. $\frac{5}{3}$ is calculated as 1). To get around this, the *floating* datatype uses some of the bits to store an *exponent*, which may be positive or negative. You throw away some of the precision of the integer representation in favor of being able to represent a much wider range of numbers.

2.3.1. 4 bytes: Floats

"Floating point" means floating decimal point—it can wash all around. You can have 16, 32 and 64 bit floats. With 32 bit floats (also called single precision float), the exponent can range from about $-38 \rightarrow +38$ and there is about 6 digits of precision. You can generate an array by specifying `dtype=np.float32` or by using exponential notation (`x=3e5`).

Printing floating point numbers to the full native precision, instead of what Python regards as "convenient", requires using a `format` statement in the `print` command. For example:

```
a= 1.23456789
print("{0:.10f}".format(a))
print("{0:.20g}".format(a))
```

prints out 10 decimal points or 20 characters respectively. Of course, we've defined `a` to higher precision than single-precision float carries, so the last bunch of numbers beyond the decimal won't be correct. To make that happen, you need…

2.3.2. 8 bytes: Double-Precision

The 64 bit float, allows the exponent to range from about $-307 \rightarrow +307$ and there is about 16 digits of precision. You can generate an array using `dtype=np.float64`. Then, when you do the following, it works:

```
a= 1.23456789
print("{0:.10f}".format(a))
```

2.4. COMPLEX NUMBERS

A complex number C consists of a pair of real numbers, one the real and one the imaginary part [in NumPy: `np.real(C)` and `np.imag(C)`]. The two numbers are always floats and can be either single- or double-precision. You generate a single-precision complex number $(2.2+i3.9)$ with: `C = np.complex64(2.2, 3.9)` and a double-precision one with `np.complex128`. The arguments can be arrays, giving you a complex-number array. There are also the complex analogues `dtype= np.complex64` and `dtype=np.complex128` that you can use in linspace to generate arrays.

For a more detailed discussion on numeric datatypes in NumPy see https://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html.

3. ORGANIZATIONAL STRUCTURES

Let us switch back from NumPy to Python and discuss some organization structures Python offers. These are only briefly described here to introduce you to the basics. You are encouraged to look up the Python documentation online which describes these in more detail and provides examples and tutorials.

3.1. TUPLES

Tuples consist of a number of values, seperated by commas. They are a little like lists but not quite the same. Tuples are non-mutable, they cannot be changed like lists. You can identify tuples by the parentheses they are enclosed by '( )', unlike lists which are enclosed in square brackets '[ ]'. Even if you never use them, you will often encounter tuples as outputs of functions of the packages and modules you may use. For example, the color of a pixel in matplotlib is stored as a tuple of (r,g,b,a) where 'a' quantifies the transperancy of the pixel.

3.2. LISTS AND ARRAYS

Lists in Python are a group of values enclosed in square brackets '[ ]'. They need not all be of the same datatype. Lists are mutable- you can add, remove values from the list dynamically. Python offers numerous methods on list objects that allows you to treat them like stacks, queues or multidimensional arrays and perform useful operations like sorting, filtering, searching etc.

Unfortunately, Python does not let you perform certain other useful operations like multiplying all the elements of a list with a scalar. For such purposes, you can either write a loop that iterates over all the elements in the list or you can use NumPy arrays. You can convert a native Python list to a NumPy array by using `np.asarray(list)`. NumPy arrays are even more flexible and allow you to compute log, cosine, sin, mean and standard deviation of arrays. You can of course multiply the array by a scalar, but you can also multiply an array with another array! Is multiplying two arrays returning a matrix multiplication or an element-wise multiplication?

3.3. DICTIONARIES

One of the most useful organizational structures Python offers is a dictionary. A dictionary is a key to value mapping. Instead of indexing by numbers (like lists), dictionaries index by keys. If you are familiar with C or IDL, these are best thought of as structures but more versatile. Dictionaries are enclosed by curly brackets '{ }' and the key:value pair are seperated by a colon. An example of a dictionary is given below:

```
data = {}
data['metadata'] = {}
data['metadata']['signal_frequency_MHz'] = 13.25
data['metadata']['sampling_frequency_MHz'] = 62.5/2
data['metadata']['number_of_blocks'] = 5
data['metadata']['samples_per_block'] = 16000
data['metadata']['voltage'] = '1V'
data['samples'] = ugradio.pico.read_socket('1V')
```

You can then store the entire dictionary above to a file! This way you will not forget what the data you stored in a particular file is representing. Can you think of more innovative ways to use dictionaries?

4. APPENDIX: HOW FLOATING-POINT NUMBERS ARE STORED IN COMPUTERS

4.1. The 32-bit IEEE standard

Here we describe the 32-bit word case and assume the IEEE convention; see *Numerical Recipes, §20.1*, for more complete information. 24 bits are reserved for the mantissa and 8 for the exponent. Each reserves one bit for the sign, leaving 23 and 7 bits for numbers.

1. For the exponent, the the maxima and minimum numbers are $\pm 2^7 = \pm 128$—or more accurately, -128 to $+127$. This exponent applies to a binary number; converting to decimal, we have $2^{128} \approx 1.7 \times 10^{38}$. Roughly, this is the maximum number that can be stored as a single-precision float.
2. For the mantissa, the maximum and minimum numbers are $\pm 2^{23} \approx 10^7$. This means that the fractional error in a stored number is about 10^{-7}. As you do more and more calculations, using the results of one in the next, this fractional error increasees.
3. For double precision, the maximum number is about 10^{308} and the precision is about 10^{-16}.