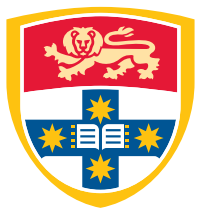
INFO1110 & COMP9001: Introduction to Programming

School of Information Technologies, University of Sydney



Copyright Warning

Week 12: More data types and Arrays

We will cover: Further understanding of memory

You should read: §§pg124-126, pg534 and Memory pg536-540

Lecture 23: Data types and simple arrays

Memory layout of types and arrays

Data types

Python has very simple types (string, int, float) and hide the memory details

We need to understand the representation of data types and their limits generally.

The data type and its representation is very important to define clearly. There will be cases that we need to be careful about which datatype we choose.

numpy is a module for python that is specifically targeting scientific applications. We will be using this module.

```
1 import numpy
2 # we have access to the numpy module using numpy.<something>
3
4 import numpy as np
5 # we have access to the numpy module using np.<something>
```

Review: Integers are stored as binary

All integers can be thought of as binary numbers: 0, 1, 10, 11, 100, 101, . . . :

base 10	**base 2 / binary**	**expansion**
0	0	0
1	1	1×2^0
2	10	$1 \times 2^1 + 0 \times 2^0$
3	11	$2^1 + 2^0$
5	101	$2^2 + 2^0$

Bit pattern to integer mapping

3 bits means $2^3 = 8$ possible values

As positive numbers, there are 8 values. Usually starting from zero.

bits	value
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Bit pattern to integer mapping (cont.)

Signed integers can be both positive and negative

Any bit pattern can be associate with any symbol. There are no real restriction of defining this mapping:

bits	value
000	-3
001	1
010	-4
011	-2
100	0
…	…

e.g. someone decided
the letter `A` will be `0100 0001`, and
the letter `a` will be `0110 0001`.

Bit pattern to integer mapping (cont.)

However, if we want arithmetic operators on bit patterns to produce results consistent with mathematical + - / *, then we need a method having both:
1) Specific bit patterns to map to signed and unsigned numbers and ;
2) Specific sequence of bitwise operations for said operators to be performed correctly.

Two's complement is the most common method.

bits	value	1st bit
000	0	positive
001	1	positive
010	2	positive
011	3	positive
100	-4	negative
101	-3	negative
110	-2	negative
111	-1	negative

Signed vs Unsigned Integers

type	bits	range
int8	8	[-128, 127]
int16	16	[-32768, 32767]
int32	32	[-2147483648, 2147483647]
int64	64	[-9223372036854775808, 9223372036854775807]
uint8	8	[0, 255]
uint16	16	[0, 65535]
uint32	32	[0, 4294967295]
uint64	64	[0, 18446744073709551615]

Signed vs Unsigned Integers (cont.)

Integer overflow occurs when we try to represent values outside the allowable range

Example of integer overflow

```
1 import numpy as np
2
3 for i in range(20):
4     x = i + 240
5     y = np.uint8(x)
6     print(y, end=' ')
7 print("")
```

```
240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 0 1 2 3
```

Signed vs Unsigned Integers (cont.)

Further example

```
1  import numpy as np
2
3  for i in range(20):
4      x = i - 10
5      y = np.uint8(x)
6      print(y, end=' ')
7  print("")
```

```
246 247 248 249 250 251 252 253 254 255 0 1 2 3 4 5 6 7 8 9
```

Review `float` type

To represent numbers using a decimal point e.g. 3.14, 0.0023, -6.21×10^3

`float` is referred to as *floating point* types.

Typical `float` type uses 4 bytes of memory, but Python 3+ has arbitrary precision.

The range and the accuracy of floating point types depends on how many bits are used in the *coefficient*, and the *exponent*. Python will manage this automatically.

Floating point number

Mapping of bit pattern to symbol
e.g. Consider the *IEEE-style* 5 bit floating point number. 1 bit sign, 2 bits exponent, 2 bits mantissa

bits	value
00010	0.5
10001	0.75
00000	+0
10000	-0
01100	Infinity
11100	-Infinity
11111	NaN
01010	2
11010	-2
00111	1.5

Floating point number (cont.)

IEEE-754 is one of the best we have so far despite the oddities around zero.

type	sign/exp/mant. bits	resolution, range
float16	1, 5, 10	0.001, ±6.55040e+04
float32	1, 8 , 23	1e-06, ±3.4028235e+38
float64	1, 11, 52	1e-15, ±1.7976931348623157e+308
float128[1]	1, 15, 63	1e-18, ±1.189731495357231765e+4932

Example of float data type

```
1 import numpy as np
2
3 x = np.float32(3.145926)
4 print(x)
```

```
3.145926
```

[1] may not be available

Errors with floating point numbers

```
1  import numpy as np
2
3  y = np.float16(1)
4  y = y / 0
5  print("still going...")
6  if np.isinf(y):
7      print("we found an infinite number, but keep going")
```

```
divzero.py:4: RuntimeWarning: divide by zero encountered in true_divide
   y = y / 0
still going...
we found an infinite number, but keep going
```

...we don't stop. Allows for error handling at a later time by the programmer.

Errors with floating point numbers (cont.)

Further errors with floating point numbers

```
1 import numpy as np
2
3 y = np.sqrt(-1)
4 print("still going...")
5 if np.isnan(y):
6     print("we found Not a Number, but keep going")
```

```
floaterror.py:11: RuntimeWarning: invalid value encountered in sqrt
   y = np.sqrt(-1)
still going...
we found Not a Number, but keep going
```

Fixed bit width data types

Why do we care about knowing fixed bit width data types?

Are you alone? Probably not.

Is the data stationary? Probably not.

Will the program run for a really long time? maybe. How long is really long?

Recall Lists

The list has been used to describe a collection of objects.

The list is a kind of data type. It is a more complicated data structure and can do further operations on collections. Automatically resize, insert in middle, delete from middle.

It relies on iterator to get from one element to the next. This is because the elements are not necessarily adjacent in memory.

Arrays

Array is the simplest form of representing a collection of a single data type in a contiguous area of memory.

Contiguous memory: that is, if we look at how bits are laid out in memory, they are all adjacent and have no gaps.

Python arrays exist, largely based on C programming language types. However, we will focus on numpy arrays and fixed width data types.

Numpy Arrays

Creating an array of integers with 32 bit width as the data type (dtype)

```
1 import numpy as np
2
3 nums = np.array([-2, 1, 55, 400], dtype=np.int32)
4 print(nums)
5 print(type(nums))
```

```
[ -2   1  55 400]
<class 'numpy.ndarray'>
```

Numpy Arrays (cont.)

Operations are the same as list index access.

```
1  import numpy as np
2
3  nums = np.array([1,2,3,4,5], dtype=np.uint8)
4  i = 0
5  while i < len(nums):
6      nums[i] += 252
7      i += 1
8  print(nums)
```

```
[253 254 255   0   1]
```

Faster construction of Numpy Arrays

Generate the array of size n

All zero values

```
1 import numpy as np
2
3 nums = np.zeros(5, dtype=np.uint8)
4 print(nums)
```

```
[0 0 0 0 0]
```

Using natural number sequence

```
1 import numpy as np
2
3 nums = np.arange(5, dtype=np.float16)
4 print(nums)
```

```
[0. 1. 2. 3. 4.]
```

Faster construction of Numpy Arrays (cont.)

Random number sequence

```
1 import numpy as np
2
3 nums = np.random.random(5).astype('float64')
4 print(nums)
```

```
[0.0815705  0.94061434 0.32178428 0.55213287 0.54824847]
```

Linearly spaced numbers, for whichever type

```
1 import numpy as np
2
3 inums = np.linspace(0, 10, 5).astype('int8')
4 print(inums)
5 fnums = np.linspace(0, 10, 5).astype('float32')
6 print(fnums)
```

```
[ 0  2  5  7 10]
[ 0.   2.5  5.   7.5 10.]
```

Further information for Numpy

Make your own types with dtype (there are no new classes to define)

Reading real data from file directly into array. Big topic, not covered. (.NPY as binary storage really fast!)

Special values for missing data `NA` are also considered with statistical packages within numpy as to not skew the results. `NaN`'s can be filtered and also replaced with other values.

Numpy is separate to Python. It is part of the Scientific Computing Tools for Python ecosystem.

Jones E, Oliphant E, Peterson P, et al. SciPy: Open Source Scientific Tools for Python, 2001-, http://www.scipy.org/ [Online; accessed 2018-05-27]