1.3.3. More elaborate arrays

Section contents

1.3.3.1. More data types

1.3.3.1.1. Casting

"Bigger" type wins in mixed-type operations:

```
>>> np.array([1, 2, 3]) + 1.5
array([ 2.5,  3.5,  4.5])
```

Assignment never changes the type!

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')
>>> a[0] = 1.9     # <-- float is truncated to integer
>>> a
array([1, 2, 3])
```

Forced casts:

```
>>> a = np.array([1.7, 1.2, 1.6])
>>> b = a.astype(int)  # <-- truncates to integer
>>> b
array([1, 1, 1])
```

Rounding:

```
>>> a = np.array([1.2, 1.5, 1.6, 2.5, 3.5, 4.5])
>>> b = np.around(a)
>>> b                    # still floating-point
array([ 1.,  2.,  2.,  2.,  4.,  4.])
>>> c = np.around(a).astype(int)
>>> c
array([1, 2, 2, 2, 4, 4])
```

1.3.3.1.2. Different data type sizes

Integers (signed):

int8	8 bits
int16	16 bits
int32	32 bits (same as **int** on 32-bit platform)
int64	64 bits (same as **int** on 64-bit platform)

```
>>> np.array([1], dtype=int).dtype
dtype('int64')
>>> np.iinfo(np.int32).max, 2**31 - 1
(2147483647, 2147483647)
```

Unsigned integers:

uint8	8 bits
uint16	16 bits
uint32	32 bits
uint64	64 bits

```
>>> np.iinfo(np.uint32).max, 2**32 - 1
(4294967295, 4294967295)
```

Floating-point numbers:

float16	16 bits
float32	32 bits
float64	64 bits (same as **float**)
float96	96 bits, platform-dependent (same as **np.longdouble**)
float128	128 bits, platform-dependent (same as **np.longdouble**)

Long integers

Python 2 has a specific type for ‘long’ integers, that cannot overflow, represented with an ‘L’ at the end. In Python 3, all integers are long, and thus cannot overflow.

```
>>> np.iinfo(np.int64).max,
    2**63 - 1
(9223372036854775807,
```

```
>>> np.finfo(np.float32).eps
1.1920929e-07
>>> np.finfo(np.float64).eps
2.2204460492503131e-16

>>> np.float32(1e-8) + np.float32(1) == 1
True
>>> np.float64(1e-8) + np.float64(1) == 1
False
```

Complex floating-point numbers:

complex64	two 32-bit floats

`complex128`	two 64-bit floats
`complex192`	two 96-bit floats, platform-dependent
`complex256`	two 128-bit floats, platform-dependent

Smaller data types

If you don't know you need special data types, then you probably don't.

Comparison on using `float32` instead of `float64`:

- Half the size in memory and on disk
- Half the memory bandwidth required (may be a bit faster in some operations)

```
In [1]: a = np.zeros((1e6,), dtype=np.float64)

In [2]: b = np.zeros((1e6,), dtype=np.float32)

In [3]: %timeit a*a
1000 loops, best of 3: 1.78 ms per loop

In [4]: %timeit b*b
1000 loops, best of 3: 1.07 ms per loop
```

- But: bigger rounding errors — sometimes in surprising places (i.e., don't use them unless you really need them)

1.3.3.2. Structured data types

`sensor_code`	(4-character string)
`position`	(float)

`value`	(float)

```
>>> samples = np.zeros((6,), dtype=[('sensor_code', 'S4'),
...                                 ('position', float), ('value', float)])
>>> samples.ndim
1
>>> samples.shape
(6,)
>>> samples.dtype.names
('sensor_code', 'position', 'value')

>>> samples[:] = [('ALFA',   1, 0.37), ('BETA', 1, 0.11), ('TAU', 1,
      0.13),
...               ('ALFA', 1.5, 0.37), ('ALFA', 3, 0.11), ('TAU', 1.2,
      0.13)]
>>> samples
array([('ALFA', 1.0, 0.37), ('BETA', 1.0, 0.11), ('TAU', 1.0, 0.13),
       ('ALFA', 1.5, 0.37), ('ALFA', 3.0, 0.11), ('TAU', 1.2, 0.13)],
      dtype=[('sensor_code', 'S4'), ('position', '<f8'), ('value', '<f8')])
```

Field access works by indexing with field names:

```
>>> samples['sensor_code']
array(['ALFA', 'BETA', 'TAU', 'ALFA', 'ALFA', 'TAU'],
      dtype='|S4')
>>> samples['value']
array([ 0.37,  0.11,  0.13,  0.37,  0.11,  0.13])
>>> samples[0]
('ALFA', 1.0, 0.37)

>>> samples[0]['sensor_code'] = 'TAU'
>>> samples[0]
('TAU', 1.0, 0.37)
```

Multiple fields at once:

```
>>> samples[['position', 'value']]
array([(1.0, 0.37), (1.0, 0.11), (1.0, 0.13), (1.5, 0.37), (3.0, 0.11),
       (1.2, 0.13)],
      dtype=[('position', '<f8'), ('value', '<f8')])
```

Fancy indexing works, as usual:

```
>>> samples[samples['sensor_code'] == 'ALFA']
array([('ALFA', 1.5, 0.37), ('ALFA', 3.0, 0.11)],
      dtype=[('sensor_code', 'S4'), ('position', '<f8'), ('value', '<f8')])
```

Note: There are a bunch of other syntaxes for constructing structured arrays, see here and here.

1.3.3.3. maskedarray: dealing with (propagation of) missing data

- For floats one could use NaN's, but masks work for all types:

```
>>> x = np.ma.array([1, 2, 3, 4], mask=[0, 1, 0, 1])
>>> x
masked_array(data = [1 -- 3 --],
             mask = [False  True False  True],
       fill_value = 999999)


>>> y = np.ma.array([1, 2, 3, 4], mask=[0, 1, 1, 1])
>>> x + y
masked_array(data = [2 -- -- --],
             mask = [False  True  True  True],
```

```
       fill_value = 999999)
```

- Masking versions of common functions:

```
>>> np.ma.sqrt([1, -1, 2, -2])
masked_array(data = [1.0 -- 1.41421356237... --],
             mask = [False  True False  True],
       fill_value = 1e+20)
```

Note: There are other useful array siblings

While it is off topic in a chapter on numpy, let's take a moment to recall good coding practice, which really do pay off in the long run:

Good practices

- Explicit variable names (no need of a comment to explain what is in the variable)
- Style: spaces after commas, around =, etc.

  A certain number of rules for writing "beautiful" code (and, more importantly, using the same conventions as everybody else!) are given in the Style Guide for Python Code and the Docstring Conventions page (to manage help strings).
- Except some rare cases, variable names and comments in English.