Understanding Data Types in Python

Effective data-driven science and computation requires understanding how data is stored and manipulated. This section outlines and contrasts how arrays of data are handled in the Python language itself, and how NumPy improves on this.

Users of Python are often drawn-in by its ease of use, one piece of which is dynamic typing. While a statically-typed language like C or Java requires each variable to be explicitly declared, a dynamically-typed language like Python skips this specification. For example, in C you might specify a particular operation as follows:

```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

While in Python the equivalent operation could be written this way:

```
# Python code
result = 0
for i in range(100):
    result += i
```

Notice the main difference: in C, the data types of each variable are explicitly declared, while in Python the types are dynamically inferred. This means, for example, that we can assign any kind of data to any variable:

```
# Python code
x = 4
x = "four"
```

Here we've switched the contents of `x` from an integer to a string. The same thing in C would lead (depending on compiler settings) to a compilation error or other unintented consequences:

```
/* C code */
int x = 4;
x = "four";  // FAILS
```

Numeric - int, float, complex

Sequence Type - string, list, tuple

Mapping Type - dict

Boolean - bool

Set Type - set

1. Numeric Data Types in Python

The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number.

In [1]:
```python
a = 5
print(type(a))

b = 5.0
print(type(b))

c = 2 + 4j
print(type(c))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

2. Sequence Data Types in Python

The sequence Data Type in Python is the ordered collection of similar or different Python data types. Sequences allow storing of multiple values in an organized and efficient fashion.

In [3]:
```python
# String data dtype
s = 'Welcome to the Data Science World!!'
```

```
print(s)

# check data type
print(type(s))

# access string with index
print(s[1])
print(s[2])
print(s[-1])
```

```
Welcome to the Data Science World!!
<class 'str'>
e
l
!
```

In [5]:
```
# List data type
# Empty list
a = []

# list with int values
a = [1, 2, 3]
print(a)

# list with mixed int and string
b = ["Python", "For", "Data Science", 4, 5]
print(b)
```

```
[1, 2, 3]
['Python', 'For', 'Data Science', 4, 5]
```

In [6]:
```
# accessing the element
print("Accessing element from the list")
print(a[0])
print(a[2])

print("Accessing element using negative indexing")
print(a[-1])
print(a[-3])
```

```
Accessing element from the list
1
3
Accessing element using negative indexing
3
1
```

In [7]:
```
# Tuple data dtype
# initiate empty tuple
tup1 = ()

tup2 = ('Python', 'For')
print("\nTuple with the use of String: ", tup2)
```

```
Tuple with the use of String:  ('Python', 'For')
```

In [8]:
```
#accessing the elements
tup1 = tuple([1, 2, 3, 4, 5])

# access tuple items
print(tup1[0])
print(tup1[-1])
print(tup1[-3])
```

```
1
5
3
```

3. Boolean Data Type in Python

Python Data type with one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false).

In [9]:
```
print(type(True))
print(type(False))
print(type(true))
```

```
<class 'bool'>
<class 'bool'>
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[9], line 3
      1 print(type(True))
      2 print(type(False))
----> 3 print(type(true))

NameError: name 'true' is not defined
```

4. Set Data Type in Python

In Python Data Types, Set is an unordered collection of data types that is iterable, mutable, and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

In [10]:
```
# initializing empty set
s1 = set()

s1 = set("PythonForDataScience")
print("Set with the use of String: ", s1)

s2 = set(["Python", "For", "Data","Science"])
print("Set with the use of List: ", s2)
```

```
Set with the use of String:  {'c', 'n', 'S', 'r', 't', 'i', 'h', 'e', 'y', 'D', 'P', 'F', 'o', 'a'}
Set with the use of List:  {'For', 'Science', 'Data', 'Python'}
```

In [13]:
```
# accessing the set elements
# loop through set
for i in s2:
    print(i, end=" ")

# check if item exist in set
print("python" in s1)
```

```
For Science Data Python False
```

5. Dictionary Data Type

A dictionary in Python is a collection of data values, used to store data values like a map, unlike other Python Data Types that hold only a single value as an element, a Dictionary holds a key: value pair.

```
In [14]: # initialize empty dictionary
d = {}


d = {1: 'Python', 2: 'For', 3: 'Data Science'}
print(d)

# creating dictionary using dict() constructor
d1 = dict({1: 'Python', 2: 'For', 3: 'Data Science'})
print(d1)
```

```
{1: 'Python', 2: 'For', 3: 'Data Science'}
{1: 'Python', 2: 'For', 3: 'Data Science'}
```

```
In [21]: # Accessing an element using key
d = {1: 'Python', 'name': 'For', 3: 'Data Science'}
print(d['name'])

# Accessing a element using get
print(d.get(1))
```

```
For
Python
```

Fixed-Type Arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in `array` module (available since Python 3.3) can be used to create dense arrays of a uniform type:

```
In [22]: import array
L = list(range(10))
A = array.array('i', L)
A
```

Out[22]:
```
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Here `'i'` is a type code indicating the contents are integers.

Much more useful, however, is the `ndarray` object of the NumPy package. While Python's `array` object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data.

In [23]:
```
import numpy as np
```

Creating Arrays from Python Lists

First, we can use `np.array` to create arrays from Python lists:

In [25]:
```
# integer array:
np.array([1, 4, 2, 5, 3])
```

Out[25]:
```
array([1, 4, 2, 5, 3])
```

Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point):

In [26]:
```
np.array([3.14, 4, 2, 3])
```

Out[26]:
```
array([3.14, 4.  , 2.  , 3.  ])
```

If we want to explicitly set the data type of the resulting array, we can use the `dtype` keyword:

In [27]:
```
np.array([1, 2, 3, 4], dtype='float32')
```

Out[27]:
```
array([1., 2., 3., 4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multi-dimensional; here's one way of initializing a multidimensional array using a list of lists:

```
In [28]: # nested lists result in multi-dimensional arrays
         np.array([range(i, i + 3) for i in [2, 4, 6]])
```

```
Out[28]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

```
In [29]: # Create a length-10 integer array filled with zeros
         np.zeros(10, dtype=int)
```

```
Out[29]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
In [30]: # Create a 3x5 floating-point array filled with ones
         np.ones((3, 5), dtype=float)
```

```
Out[30]: array([[1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1.]])
```

```
In [31]: # Create a 3x5 array filled with 3.14
         np.full((3, 5), 3.14)
```

```
Out[31]: array([[3.14, 3.14, 3.14, 3.14, 3.14],
                [3.14, 3.14, 3.14, 3.14, 3.14],
                [3.14, 3.14, 3.14, 3.14, 3.14]])
```

```
In [32]: # Create an array filled with a linear sequence
         # Starting at 0, ending at 20, stepping by 2
         # (this is similar to the built-in range() function)
         np.arange(0, 20, 2)
```

```
Out[32]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
In [33]: # Create an array of five values evenly spaced between 0 and 1
         np.linspace(0, 1, 5)
```

```
Out[33]: array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
In [34]: # Create a 3x3 array of uniformly distributed
         # random values between 0 and 1
         np.random.random((3, 3))
```

```
Out[34]: array([[0.97396274, 0.3671068 , 0.90825645],
                [0.0671108 , 0.60622344, 0.42896673],
                [0.44812659, 0.44348227, 0.83858516]])
```

```
In [35]: # Create a 3x3 array of normally distributed random values
         # with mean 0 and standard deviation 1
         np.random.normal(0, 1, (3, 3))
```

```
Out[35]: array([[ 0.38322411,  0.88213049, -1.0579549 ],
                [ 0.46545123,  0.46393174, -0.62647453],
                [-0.06419708,  1.16518106, -0.01177581]])
```

```
In [36]: # Create a 3x3 array of random integers in the interval [0, 10)
         np.random.randint(0, 10, (3, 3))
```

```
Out[36]: array([[5, 4, 3],
                [2, 3, 6],
                [0, 0, 7]])
```

```
In [37]: # Create a 3x3 identity matrix
         np.eye(3)
```

```
Out[37]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])
```

```
In [38]: # Create an uninitialized array of three integers
         # The values will be whatever happens to already exist at that memory location
         np.empty(3)
```

```
Out[38]: array([1., 1., 1.])
```

NumPy Standard Data Types

NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard NumPy data types are listed in the following table. Note that when constructing an array, they can be specified using a string:

```
np.zeros(10, dtype='int16')
```

Or using the associated NumPy object:

```
np.zeros(10, dtype=np.int16)
```

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)

Data type	Description
`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