Understanding Python's pandas source code

D. Gueorguiev 5/26/21

https://github.com/pandas-dev/pandas

Contents

Prerequisite

The reader is assumed to have working knowledge of python v3 (Pandas source code), C (CPython C-API, NumPy, Swig source code) and C++ (Google Flatbuffers, Apache Arrow and Cpp-Parquet source code).

Preliminaries and third-party packages

Python C API Basics

Some Python C-API preliminaries:
Note: The discussion in this paragraph is inspired by https://tenthousandmeters.com/blog/python-behind-the-scenes-6-how-python-object-system-works/.
There are large number of structures which are used in the definition of object types for Python. Below are discussed some base object types and macros.
All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the PyObject and PyVarObject types, which are defined, in turn, by the expansions of some macros also used, whether directly or undirectly, in the definition of all other Python objects.

PyObject: all object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In "normal" release build it contains only the object reference count and a pointer to the corresponding type object. Nothing is actually declared to be a `PyObject` but every pointer to a Python object can be cast to `PyObject*`. Access to members must be done by using the macros `Py_REFCNT` and `Py_TYPE`. `Py_REFCNT(o)` is used to access the

ob_refcnt member of a Python object. It expands to: (((PyObject*)(o))->ob_refcnt). Py_TYPE(o) is used to access the ob_type member of a Python object and expands to (((PyObject*)(o))->ob_type).
PyVarObject: this is an extension of PyObject that adds the ob_size field. This is only used for objects which have some notion of length. This type does not appear often in Python C API. Access to the members must be done by using the macros Py_REFCNT, Py_TYPE, and Py_SIZE. Py_SIZE(o) accesses the ob_size member of a Python object and it expands to (((PyVarObject*)(o))->ob_size).

PyObject_HEAD: this is a macro used when declaring new types which represent objects without a varying length. The PyObject_HEAD macro expands to PyObject ob_base;.
PyObject_VAR_HEAD: this is a macro used when declaring new types which represent objects with a length which varies from instance to instance. This macro expands to PyVarObject ob_base;.
Some useful typedefs in Include/object.h:

```
typedef PyObject * (*unaryfunc)(PyObject *);
typedef PyObject * (*binaryfunc)(PyObject *, PyObject *);
typedef PyObject * (*ternaryfunc)(PyObject *, PyObject *, PyObject *);
typedef int (*inquiry)(PyObject *);
typedef void (*destructor)(PyObject *);
typedef PyObject *(*getattrfunc)(PyObject *, char *);
typedef int (*setattrfunc)(PyObject *, char *, PyObject *);

typedef ssize_t         Py_ssize_t; /* Include/pyport.h */
```

PyObject is defined as:

```
typedef struct _object {
    _PyObject_HEAD_EXTRA // macro, for debugging purposes only
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
} PyObject;
```

It has two members :

- A reference count ob_refcnt that CPython uses for garbage collection
- A pointer to the object's type ob_type;

The Python VM treats any Python object as PyObject. This is achieved by "extending" PyObject – defining a new structure which first member is PyObject. This trick relies on the C standard according to which any pointer to a C struct can be converted to the pointer of its first object. Here is for example how the float object is defined:

```
typedef struct {
    PyObject ob_base; // expansion of PyObject_HEAD macro
    double ob_fval;
} PyFloatObject;
```

A float object stores everything PyObject stores plus a floating point value ob_fval. Obviously we could do the following

```
PyFloatObject float_object;
// ...
PyObject *obj_ptr = (PyObject *)&float_object;
PyFloatObject *float_obj_ptr = (PyFloatObject *)obj_ptr;
```

The reason why the Python VM treats every object as PyObject is because it needs to access the object's type. A Python type is also an object, an instance of PyTypeObject struct.

Type Objects
One of the most important structures in the Python Object system is the structure that defines a new type: the PyTypeObject structure.

```
typedef struct _typeobject PyTypeObject;
. . .
// If this structure is modified, Doc/includes/typestruct.h should be updated
```

```
// as well.
struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a static type
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
```

```
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6 */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;
};
```

Notice the first member of PyTypeObject which is PyVarObject and not PyObject. This is dictated by the PyTypeObject's need to implement the notion of length. So, what is a type and why does it have so many members? A type determines how the objects of this type behave. Each member of a type called a slot is responsible for a particular aspect of the object's behavior. For example:

`tp_new` is a pointer to a function that creates new objects of the type
`tp_str` is a pointer to a function that implements `str()` for objects of the type
`tp_hash` is a pointer to a function that implements `hash()` for objects of the type

Some slots called sub-slots are grouped together in suites. A suite is just a struct, which contains related slots. For example, `PySequenceMethods` struct is a suite of subslots which implement the sequence protocol:

```
typedef struct {
    lenfunc sq_length;
    binaryfunc sq_concat;
    ssizeargfunc sq_repeat;
    ssizeargfunc sq_item;
    void *was_sq_slice;
    ssizeobjargproc sq_ass_item;
    void *was_sq_ass_slice;
    objobjproc sq_contains;

    binaryfunc sq_inplace_concat;
    ssizeargfunc sq_inplace_repeat;
} PySequenceMethods;
```

All type objects available at the latest version of the Python 3 type system are documented here. As an example let us investigate how Python add objects by looking into the type objects hierarchy. All slots that deal with arithmetic operations on numbers are stored in the `PyNumberMethods` structure.

`PyNumberMethods`: this structure holds pointers to the functions which an object uses to implement the number protocol. Each function is used by the function of similar name documented in the Number Protocol. Here is how it looks like:

```
typedef struct {
    /* Number implementations must check *both*
       arguments for proper type and implement the necessary conversions
       in the slot functions themselves. */

    binaryfunc nb_add;  // typedef PyObject * (*binaryfunc)(PyObject *, PyObject *)
    binaryfunc nb_subtract;
    binaryfunc nb_multiply;
    binaryfunc nb_remainder;
    binaryfunc nb_divmod;
    ternaryfunc nb_power;
    unaryfunc nb_negative;
    unaryfunc nb_positive;
    unaryfunc nb_absolute;
    inquiry nb_bool;
```

```
    unaryfunc nb_invert;
    binaryfunc nb_lshift;
    binaryfunc nb_rshift;
    binaryfunc nb_and;
    binaryfunc nb_xor;
   // ... more sub-slots
} PyNumberMethods;
```

The first member of this suite is a binary function named nb_add. The Python VM calls nb_add to execute the BINARY_ADD opcode. Let us take a look how BINARY_ADD is implemented in Python/ceval.c:

```
case TARGET(BINARY_ADD): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum;
    /* NOTE(vstinner): Please don't try to micro-optimize int+int on
        CPython using bytecode, it is simply worthless.
        See http://bugs.python.org/issue21955 and
        http://bugs.python.org/issue10044 for the discussion. In short,
        no patch shown any impact on a realistic benchmark, only a minor
        speedup on microbenchmarks. */
    if (PyUnicode_CheckExact(left) &&
                PyUnicode_CheckExact(right)) {
        sum = unicode_concatenate(tstate, left, right, f, next_instr);
        /* unicode_concatenate consumed the ref to left */
    }
    else {
        sum = PyNumber_Add(left, right);
        Py_DECREF(left);
    }
    Py_DECREF(right);
    SET_TOP(sum);
    if (sum == NULL)
        goto error;
    DISPATCH();
}
```

Notice the invocation of unicode_concatenate with the arguments left and right. This happens only if the object types of left and right both are instances of PyUnicode_Type because PyUnicode_CheckExact is defined as:

```
PyAPI_DATA(PyTypeObject) PyUnicode_Type;
./Include/unicodeobject.h:116: #define PyUnicode_CheckExact(op) Py_IS_TYPE(op, &PyUnicode_Type)
```

So if the objects left and right are strings we call unicode_concatenate instead of PyNumber_Add. This is a kind of optimization as unicode_concatenate would mutate left if it has no other references. On may want to introduce additional optimizations but the comment by the developer vstinner in the excerpt above warns against it as every additional optimization is useful only in case it is executed otherwise it would introduce additional performance overhead.

Let us look now into PyNumber_Add given with the excerpt below:

```
PyObject *
PyNumber_Add(PyObject *v, PyObject *w)
{
    PyObject *result = BINARY_OP1(v, w, NB_SLOT(nb_add), "+");
    if (result != Py_NotImplemented) {
        return result;
    }
    Py_DECREF(result);

    PySequenceMethods *m = Py_TYPE(v)->tp_as_sequence;
    if (m && m->sq_concat) {
        result = (*m->sq_concat)(v, w);
        assert(_Py_CheckSlotResult(v, "+", result != NULL));
        return result;
    }
```

```
    return binop_type_error(v, w, "+");
}
```

`binary_op1` is given with the following excerpt (Objects/abstract.c:777):

```
static PyObject *
binary_op1(PyObject *v, PyObject *w, const int op_slot
#ifndef NDEBUG
           , const char *op_name
#endif
           )
{
    binaryfunc slotv;
    if (Py_TYPE(v)->tp_as_number != NULL) {
        slotv = NB_BINOP(Py_TYPE(v)->tp_as_number, op_slot);
    }
    else {
        slotv = NULL;
    }

    binaryfunc slotw;
    if (!Py_IS_TYPE(w, Py_TYPE(v)) && Py_TYPE(w)->tp_as_number != NULL) {
        slotw = NB_BINOP(Py_TYPE(w)->tp_as_number, op_slot);
        if (slotw == slotv) {
            slotw = NULL;
        }
    }
    else {
        slotw = NULL;
    }

    if (slotv) {
        PyObject *x;
        if (slotw && PyType_IsSubtype(Py_TYPE(w), Py_TYPE(v))) {
            x = slotw(v, w);
            if (x != Py_NotImplemented)
                return x;
            Py_DECREF(x); /* can't do it */
            slotw = NULL;
        }
        x = slotv(v, w);
        assert(_Py_CheckSlotResult(v, op_name, x != NULL));
        if (x != Py_NotImplemented) {
            return x;
        }
        Py_DECREF(x); /* can't do it */
    }
    if (slotw) {
        PyObject *x = slotw(v, w);
        assert(_Py_CheckSlotResult(w, op_name, x != NULL));
        if (x != Py_NotImplemented) {
            return x;
        }
        Py_DECREF(x); /* can't do it */
    }
    Py_RETURN_NOTIMPLEMENTED;
}
```

Here the macro is defined in Objects/abstract.c:762 as:

```
#define NB_BINOP(nb_methods, slot) \
        (*(binaryfunc*)(& ((char*)nb_methods)[slot]))
```

The `binary_op1` function takes three parameters: the left operand `v`, the right operand `w` and an offset `op_slot` that identifies the slot. Types of both operands can implement the slot. Therefore, binary_op1() looks up both implementations. To calculate the result, it calls one implementation or another relying on the following fact:

1. If the type of the operand is a subtype of another, call the slot of the subtype
2. If the left operand does not have the slot, call the slot of the right operand

3. Otherwise call the slot of the left operand.

The reason to prioritize the slot of a subtype is to allow subtypes to override the behavior of their ancestors:

```
>>> class HungryInt(int):
...     def __add__(self, o):
...             return self
...
>>> x = HungryInt(5)
>>> x + 2
5
>>> 2 + x
7
>>> HungryInt.__radd__ = lambda self, o: self
>>> 2 + x
5
```

The function `PyType_isSubtype` is defined in ./Objects/typeobject.c:1524 as:

```
int
PyType_IsSubtype(PyTypeObject *a, PyTypeObject *b)
{
    PyObject *mro;

    mro = a->tp_mro;
    if (mro != NULL) {
        /* Deal with multiple inheritance without recursion
           by walking the MRO tuple */
        Py_ssize_t i, n;
        assert(PyTuple_Check(mro));
        n = PyTuple_GET_SIZE(mro);
        for (i = 0; i < n; i++) {
            if (PyTuple_GET_ITEM(mro, i) == (PyObject *)b)
                return 1;
        }
        return 0;
    }
    else
        /* a is not completely initialized yet; follow tp_base */
        return type_is_subtype_base_chain(a, b);
}
```

In ./Objects/typeobject.c:1511 :

```
/* type test with subclassing support */

static int
type_is_subtype_base_chain(PyTypeObject *a, PyTypeObject *b)
{
    do {
        if (a == b)
            return 1;
        a = a->tp_base;
    } while (a != NULL);

    return (b == &PyBaseObject_Type);
}
```

If `binary_op1()` succeeds then `PyNumber_Add()` simply returns the result of `binary_op1()`. If, however, `binary_op1()` returns the `NotImplemented` constant which means that the operation cannot be performed for a given combination of types, `PyNumber_Add()` calls the `sq_concat` sequence slot of the first operand and returns the result of this call:

```
    PySequenceMethods *m = Py_TYPE(v)->tp_as_sequence;
    if (m && m->sq_concat) {
        result = (*m->sq_concat)(v, w);
        assert(_Py_CheckSlotResult(v, "+", result != NULL));
        return result;
    }
```

A type can support the + operator either by implementing `nb_add` or `sq_concat`. These slots have different meanings:

- `nb_add` means algebraic notation with properties like `a + b = b + a`
- `sq_concat` means concatenation of sequences.

Built-in types such as `int` and `float` implement `nb_add` and built-in types such as `str` and `list` implement `sq_concat`. `sq_concat` is set to `NULL` for all user-defined types.

What `nb_add` is set to?
Since addition is a different operation for different types the `nb_add` slot of a type must be one of two things:

- it is either type-specific function that adds object of that type; or
- it is a type-agnostic function that calls some type-specific functions such as the type's `__add__()` special method.

Which one of the two `nb_add` slot is depends on the type. For example, built-in types such as int and float have their own implementation of nb_add. In contrast, all classes share the same implementation. Fundamentally, built-in types and user-defined classes are instances of PyTypeObject. The difference between those is how they are created – this difference effects the way the slots are set.

Ways to create a type:
There are two ways to create a type object –

- by statically defining it or
- by dynamically allocating it

Take for example the built-in float type defined in ./Objects/floatobject.c:1928:

```
PyTypeObject PyFloat_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "float",
    sizeof(PyFloatObject),
    0,
    (destructor)float_dealloc,                  /* tp_dealloc */
    0,                                          /* tp_vectorcall_offset */
    0,                                          /* tp_getattr */
    0,                                          /* tp_setattr */
    0,                                          /* tp_as_async */
    (reprfunc)float_repr,                       /* tp_repr */
    &float_as_number,                           /* tp_as_number */
    0,                                          /* tp_as_sequence */
    0,                                          /* tp_as_mapping */
    (hashfunc)float_hash,                       /* tp_hash */
    0,                                          /* tp_call */
    0,                                          /* tp_str */
    PyObject_GenericGetAttr,                    /* tp_getattro */
    0,                                          /* tp_setattro */
    0,                                          /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
        _Py_TPFLAGS_MATCH_SELF,                 /* tp_flags */
    float_new__doc__,                           /* tp_doc */
    0,                                          /* tp_traverse */
    0,                                          /* tp_clear */
    float_richcompare,                          /* tp_richcompare */
    0,                                          /* tp_weaklistoffset */
    0,                                          /* tp_iter */
    0,                                          /* tp_iternext */
    float_methods,                              /* tp_methods */
    0,                                          /* tp_members */
    float_getset,                               /* tp_getset */
    0,                                          /* tp_base */
    0,                                          /* tp_dict */
    0,                                          /* tp_descr_get */
    0,                                          /* tp_descr_set */
    0,                                          /* tp_dictoffset */
    0,                                          /* tp_init */
    0,                                          /* tp_alloc */
    float_new,                                  /* tp_new */
    .tp_vectorcall = (vectorcallfunc)float_vectorcall,
```

```
};
```

The slots of the statically defined types are specified explicitly. We can easily see how the float type implements nb_add by looking at the "number" suite `float_as_number` in ./Objects/floatobject.c:1892

```
static PyNumberMethods float_as_number = {
    float_add,          /* nb_add */
    float_sub,          /* nb_subtract */
    float_mul,          /* nb_multiply */
    float_rem,          /* nb_remainder */
    float_divmod,       /* nb_divmod */
    float_pow,          /* nb_power */
    (unaryfunc)float_neg, /* nb_negative */
    float_float,        /* nb_positive */
    (unaryfunc)float_abs, /* nb_absolute */
    (inquiry)float_bool, /* nb_bool */
    0,                  /* nb_invert */
    0,                  /* nb_lshift */
    0,                  /* nb_rshift */
    0,                  /* nb_and */
    0,                  /* nb_xor */
    0,                  /* nb_or */
    float___trunc___impl, /* nb_int */
    0,                  /* nb_reserved */
    float_float,        /* nb_float */
    0,                  /* nb_inplace_add */
    0,                  /* nb_inplace_subtract */
    0,                  /* nb_inplace_multiply */
    0,                  /* nb_inplace_remainder */
    0,                  /* nb_inplace_power */
    0,                  /* nb_inplace_lshift */
    0,                  /* nb_inplace_rshift */
    0,                  /* nb_inplace_and */
    0,                  /* nb_inplace_xor */
    0,                  /* nb_inplace_or */
    float_floor_div,    /* nb_floor_divide */
    float_div,          /* nb_true_divide */
    0,                  /* nb_inplace_floor_divide */
    0,                  /* nb_inplace_true_divide */
};
```

where we find the `float_add()` function in ./Objects/floatobject.c:562, a straightforward implementation of `nb_add`:

```
static PyObject *
float_add(PyObject *v, PyObject *w)
{
    double a,b;
    CONVERT_TO_DOUBLE(v, a);
    CONVERT_TO_DOUBLE(w, b);
    a = a + b;
    return PyFloat_FromDouble(a);
}
```

This example demonstrated how to specify the behavior of statically defined type which is straightforward – just write the implementations of the relevant slots and point each slot to the corresponding implementation.

Defining Extension Types

Note: The discussion on this paragraph comes from https://docs.python.org/3/extending/newtypes_tutorial.html

Python allows the writer of C extension module to define new types which can be manipulated from Python code, much like the built-in `list` and `str` types.

The Basics Of Defining Extension Types

Recall, the CPython runtime sees all Python objects as variables of type PyObject* , which is the "base type" for all Python objects. As we have seen the PyObject struct only contains the object's reference count and a pointer to the object's type object. This is where the action is – the type object determines which C functions get called by the interpreter when, for instance, an attribute gets looked up on an object, a method called, or it is multiplied by another object. So if we want to define an extension type we need to create a new type object.

The traditional way for defining static extension types is shown below. The C-API allows also defining heap-allocated extension types using the PyType_FromSpec().

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

typedef struct {
    PyObject_HEAD
    /* Type-specific fields go here. */
} CustomObject;

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

This code excerpt defines three things:

- what a `Custom` object contains – this is the `CustomObject` struct which is allocated once per each Custom instance.
- How the Custom type behaves : this is the CustomType struct which defines a set of flags and function pointers that the interpreter inspects when specific operations are requested.
- How to initialize the custom module – this is the PyInit_custom function and the associated custommodule struct

//TODO:

PySequenceMethods, PyMappingMethods, PySendResult, PyAsyncMethods, PyBufferProcs:

```
typedef struct {
    lenfunc sq_length;
    binaryfunc sq_concat;
    ssizeargfunc sq_repeat;
    ssizeargfunc sq_item;
```

```
    void *was_sq_slice;
    ssizeobjargproc sq_ass_item;
    void *was_sq_ass_slice;
    objobjproc sq_contains;

    binaryfunc sq_inplace_concat;
    ssizeargfunc sq_inplace_repeat;
} PySequenceMethods;

typedef struct {
    lenfunc mp_length;
    binaryfunc mp_subscript;
    objobjargproc mp_ass_subscript;
} PyMappingMethods;

typedef PySendResult (*sendfunc)(PyObject *iter, PyObject *value, PyObject **result);

typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
    sendfunc am_send;
} PyAsyncMethods;

typedef struct {
     getbufferproc bf_getbuffer;
     releasebufferproc bf_releasebuffer;
} PyBufferProcs;
```

NumPy

https://github.com/numpy/numpy
https://numpy.org/doc/stable/user/

NumPy Basics

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices) and an assortment of routines for fast operations on arrays including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulations, etc.

At the core of the NumPy package is the ndarray object. This encapsulates n-dimensional arrays of homogenenous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have fixed size at creation, unlike Python lists which can grow dynamically. Changing the size of an ndarray will create a new array and delete the original
- The elements in a NumPy array are all required to be of the same data type , and thus will be the same size in memory. The exception is that one can have arrays of (Python including NumPy) objects thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than it is possible using Python built-in sequences
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software just knowing how to use Python sequences is insufficient – one also need to know how to use NumPy arrays.

The points about sequence size and speed are particularly important in scientific computing. As a simple example, consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data is stored in two Python lists, **a** and **b**, we could iterate over each element as:

```
c = []
for i in range(len(a)):
   c.append(a[i]*b[i])
```

This produces the correct answer but if **a** and **b** each contain millions of numbers, we will pay the price for inefficiencies of looping in Python. We could achieve the same task much more quickly in C by writing (for clarity we neglect variable declarations, initializations, memory allocations):

```
for (i = 0; i < rows; i++) {
    c[i] = a[i]*b[i];
}
```

This saves all the overhead involved in interpreting Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python. Furthermore, the coding work required increases with the dimensionality of the data. In the case of 2-D array, for example, the C code abridged as before expands to:

```
for (i = 0; i < rows; i++)  {
    for (j = 0; j < columns; j++) {
        c[i][j] = a[i][j]*b[i][j];
    }
}
```

NumPy gives us the best of both worlds: element-by-element operations are the "default mode" when ndarray is involved, but element-by-element operation is speedily executed by pre-compiled C code. In NumPy:

```
c = a * b
```

does what the earlier examples do in near C speeds, but with code simplicity we expect from something based on Python. Indeed, the numpy idiom is even simpler. This last example illustrates two of NumPy's features which are basis of its power – broadcasting and vectorization. Vectorization describes the absence of any explicit looping and indexing in the code. Looping and indexing are taking place behind the scenes in optimized pre-compiled C code. Vectorized code has many advantages among which are:

- Vectorized code is more concise and easier to read
- Fewer lines of code means fewer bugs
- The code closely resembles standard mathematical notation hence is more suitable for scientific computations

Broadcasting is the term used to describe the implicit element-by-element behavior of operations. In NumPy all operations not just arithmetic operations but logical, bit-wise, functional behave in this implicit element-by-element fashion i.e. they broadcast. Moreover, **a** and **b** could be multidimensional arrays of the same shape or a scalar and an array or even two arrays with different shapes provided that the smaller array is expandable to the shape of the larger such that the broadcast is unambiguous.

Data Types

Array Types and conversion between types

The primitive types supported are closely tied to those in C:

NumPy type	C type	Description
numpy.bool_	bool	Boolean (True or False) stored as a byte
numpy.byte	signed char	Platform defined
numpy.ubyte	unsigned char	Platform defined
numpy.short	short	Platform defined
numpy.ushort	unsigned short	Platform defined
numpy.intc	int	Platform defined
numpy.uintc	unsigned int	Platform defined
numpy.int_	long	Platform defined
numpy.uint	unsigned long	Platform defined
numpy.longlong	long long	Platform defined
numpy.ulonglong	unsigned long long	Platform defined
numpy.half, numpy.float16		Half precision float, sign bit, 5 bits exponent, 10 bits mantissa
numpy.single	float	Platform defined single precision float: typically sign bit 8 bits exponent, 23 bit mantissa
numpy.double	double	Platform defined double precision float: typically sign bit, 11 bits exponent, 52 bit mantissa
numpy.longdouble	long double	Platform-defined extended precision float

numpy.csingle	float complex	Complex number represented by two single precision floats
numpy.cdouble	double complex	Complex number represented by two double precision floats
numpy.clongdouble	long double complex	Complex number represented by two extended precision floats

Since many of these types have platform-dependent definitions a set of fixed-size aliases are provided:

Numpy type	C type	Description
numpy.int8	int8_t	Byte (-128 to 127)
numpy.int16	int16_t	Integer (-32768 to 32767)
numpy.int32	int32_t	Integer (-2147483648 to 2147483647)
numpy.int64	int64_t	Integer (-9223372036854775808 to 9223372036854775807)
numpy.uint8	uint8_t	Unsigned Integer (0 to 255)
numpy.uint16	uint16_t	Unsigned Integer (0 to 65535)
numpy.uint32	uint32_t	Unsigned Integer (0 to 4294967295)

NumPy Source Code discussion

New Python types and C-Structs defined by NumPy in C code
Several new types are defined in C code. Most of these are accessible from Python but few are not exposed due to their limited use. Every new Python type has an associated `PyObject*` with an internal structure that includes a pointer to a "method table" that defines how the new object behaves in Python. When you receive a Python object into C code you always get a pointer to a `PyObject` structure

Numpy/arrayscalars.h

```
#ifndef _MULTIARRAYMODULE
    typedef struct {
        PyObject_HEAD
        npy_bool obval;
    } PyBoolScalarObject;
#endif


    typedef struct {
        PyObject_HEAD
        signed char obval;
    } PyByteScalarObject;


    typedef struct {
        PyObject_HEAD
        short obval;
    } PyShortScalarObject;


    typedef struct {
        PyObject_HEAD
        int obval;
    } PyIntScalarObject;


    typedef struct {
        PyObject_HEAD
        long obval;
    } PyLongScalarObject;


    typedef struct {
```

```
    PyObject_HEAD
    npy_longlong obval;
} PyLongLongScalarObject;


typedef struct {
    PyObject_HEAD
    unsigned char obval;
} PyUByteScalarObject;


typedef struct {
    PyObject_HEAD
    unsigned short obval;
} PyUShortScalarObject;


typedef struct {
    PyObject_HEAD
    unsigned int obval;
} PyUIntScalarObject;


typedef struct {
    PyObject_HEAD
    unsigned long obval;
} PyULongScalarObject;


typedef struct {
    PyObject_HEAD
    npy_ulonglong obval;
} PyULongLongScalarObject;


typedef struct {
    PyObject_HEAD
    npy_half obval;
} PyHalfScalarObject;


typedef struct {
    PyObject_HEAD
    float obval;
} PyFloatScalarObject;


typedef struct {
    PyObject_HEAD
    double obval;
} PyDoubleScalarObject;


typedef struct {
    PyObject_HEAD
    npy_longdouble obval;
} PyLongDoubleScalarObject;
```

```
typedef struct {
        PyObject_HEAD
        npy_cfloat obval;
} PyCFloatScalarObject;


typedef struct {
        PyObject_HEAD
        npy_cdouble obval;
} PyCDoubleScalarObject;


typedef struct {
        PyObject_HEAD
        npy_clongdouble obval;
} PyCLongDoubleScalarObject;


typedef struct {
        PyObject_HEAD
        PyObject * obval;
} PyObjectScalarObject;

typedef struct {
        PyObject_HEAD
        npy_datetime obval;
        PyArray_DatetimeMetaData obmeta;
} PyDatetimeScalarObject;

typedef struct {
        PyObject_HEAD
        npy_timedelta obval;
        PyArray_DatetimeMetaData obmeta;
} PyTimedeltaScalarObject;


typedef struct {
        PyObject_HEAD
        char obval;
} PyScalarObject;

#define PyStringScalarObject PyStringObject
#define PyUnicodeScalarObject PyUnicodeObject

typedef struct {
        PyObject_VAR_HEAD
        char *obval;
        PyArray_Descr *descr;
        int flags;
        PyObject *base;
} PyVoidScalarObject;

/* Macros
    Py<Cls><bitsize>ScalarObject
    Py<Cls><bitsize>ArrType_Type
  are defined in ndarrayobject.h
*/
```

```
#define PyArrayScalar_False ((PyObject *)(&(_PyArrayScalar_BoolValues[0])))
#define PyArrayScalar_True ((PyObject *)(&(_PyArrayScalar_BoolValues[1])))
#define PyArrayScalar_FromLong(i) \
    ((PyObject *)(&(_PyArrayScalar_BoolValues[((i)!=0)])))
#define PyArrayScalar_RETURN_BOOL_FROM_LONG(i)               \
    return Py_INCREF(PyArrayScalar_FromLong(i)), \
           PyArrayScalar_FromLong(i)
#define PyArrayScalar_RETURN_FALSE           \
    return Py_INCREF(PyArrayScalar_False),  \
           PyArrayScalar_False
#define PyArrayScalar_RETURN_TRUE            \
    return Py_INCREF(PyArrayScalar_True),   \
           PyArrayScalar_True

#define PyArrayScalar_New(cls) \
    Py##cls##ArrType_Type.tp_alloc(&Py##cls##ArrType_Type, 0)
#define PyArrayScalar_VAL(obj, cls)          \
    ((Py##cls##ScalarObject *)obj)->obval
#define PyArrayScalar_ASSIGN(obj, cls, val) \
    PyArrayScalar_VAL(obj, cls) = val
```

Google Flatbuffers

https://google.github.io/flatbuffers/
https://github.com/google/flatbuffers

From Flatbuffers Programmers Guide https://google.github.io/flatbuffers/usergroup0.html

Flatbuffers is efficient serialization library supporting various languages.
Benefits of using Flatbuffers:

- Access to serialized data without parsing/unpacking – what sets Flatbuffers apart is that it represents hierarchical data in a flat binary buffer in such way that it still can be accessed without parsing / unpacking while also still supporting data structure evolution (forward/backward compatibility)
- Memory efficiency and speed- the only memory needed to access your data. It requires 0 additional allocations in C++. Flatbuffers is also very suitable for use of mmap (or streaming) requiring only a part of the buffer to be in memory. Access is close to the speed of raw struct access with only one extra indirection (a kind of vtable) to allow for format evolution and optional fields. It is aimed at projects where spending time and space (many memory allocations) to be able to access or construct serialized data is undesirable such as in games or any other performance sensitive applications
- Flexible – Optional fields means not only do you get forwards and backwards compatibility (do not have to update all data with each new version). It also means you have a lot of choice in what data you write and what data you don't and how you design data structures.
- Tiny code footprint – small amounts of generated code and just a single small header as the minimum dependency which is very easy to integrate.
- Strongly typed – Errors happen at compile time rather than manually having to write repetitive and error prone run-time checks. Useful code can be generated for you.
- Convenient to use – generated C++ code allows for terse access and construction code.
- Cross platform code with no dependencies

How to use Flatbuffers:

- Write a schema file that allows you to define the data structures you may want to serialize. Fields can have scalar types (int / floats of all sizes) or they can be : string, array of any type; reference to yet another object; or a set of possible objects (unions). Fields are optional and have defaults so they do not need to be present for every object instance.
- Use `flatc` (the FlatBuffer compiler) to generate C++ header (or Java/Kotlin/C#/Go/Python classes) with helper classes to access and construct the serialized data. This header (say mydata_generated.h) only depends on flatbuffers.h which defines the core functionality.

- User the `FlatBufferBuilder` class to construct a flat binary buffer. The generated functions allow you to add objects to this buffer recursively often as simply as making a single function call.
- Store or send the buffer somewhere
- When reading it back you can obtain a pointer to the root object from the binary buffer and from there traverse it conveniently in place with `object->field()`.

Small example:

Writing the Monster's FlatBuffer Schema

To start working with FlatBuffers you first need to create a schema file which defines the format of each schema file you wish to serialize. Here is the schema that defines the template for our monsters:

```
// Example IDL file for our monster's schema
namespace MyGame.Sample;

enum Color:byte { Red = 0, Green, Blue = 2}

union Equipment { Weapon }   // optionally add more tables

struct Vec3 {
   x:float;
   y:float;
   z:float;
}

table Monster {
   pos:Vec3; // Struct
   mana:short = 150;
   hp:short = 100;
   name:string;
   friendly:bool = false (deprecated);
   inventory:[ubyte]; // Vector of scalars
   color:Color = Blue; // Enum
   weapons:[Weapon];  // Vector of tables
   equipped:Equipment; // Union
   path:[Vec3]; // Vector of structs
}
table Weapon {
   name:string;
   damage:short;
}

root_type Monster;
```

The `schema` starts with a `namespace` declaration. This determines the corresponding package/namespace for the generated code. In our example we have `Sample` namespace inside the `MyGame` namespace. Next we have an `enum` definition. In this example, we have `enum` of type `byte`, named `Color`. We have three values in this `enum`: `Red` , `Green` and `Blue`. We specify `Red = 0` and `Blue = 2`, but we do not specify an explicit value for `Green`. Since the behavior of an `enum` is to increment if unspecified `Green` will receive the implicit value of 1. Following the `enum` is a `union`. The `union` in this example is not very useful as it only contains the one `table` (named `Weapon`). If we had created multiple tables that we would want the `union` to be able to reference we could add more elements to the `union Equipment`. After the `union` comes a `struct Vec3`, which represents a floating point vector with *3* dimensions. We use `struct` here instead of `table` as `structs` are ideal for modeling data structures which will not change since they use less memory and have faster lookup. The `Monster` table is the main object in our FlatBuffer. This will be used as a template to store our `orc` monster. We specify some default values for fields such as `mana:short = 150`. If unspecified, scalar fields like `int`, `uint` or `float` will be given a default of 0 while strings and tables will be given a default of `null`. Another thing to note is the line `friendly: bool = false (deprecated);` . Since you cannot delete fields from a table (to support backwards compatibility), you can set fields as `deprecated`, which will prevent the generation of accessors for this field in the generated code. The keyword `deprecated` can break legacy code that used that accessor. The `Weapon` table is a sub-table used within our `FlatBuffer`. It is used twice: once within the `Monster` table and once within the `Equipment` union. For our `Monster` it is used to populate a vector of tables via the weapons field within our `Monster`. It is also the only table referenced by the `Equipment` union. The last part of the `schema` is the `root_type`. The root type declares what will be the root table for the serialized data. In our case the root type is our `Monster` table. The scalar types can also use alias type names such as `int16` instead of `short` and `float32` instead of `float`. Thus we could also write the `Weapon` table as:

```
table Weapon {
   name:string;
   damage:int16;
}
```

More information about Flatbuffers Schema
Let us look again into the same example of Flatbuffers schema with small modifications:

```
// example IDL file
namespace MyGame;

attribute “priority”;

enum Color : byte { Red = 1; Green; Blue }

union Any { Monster, Weapon, Pickup }

struct Vec3 {
   x:float;
   y:float;
   z:float;
}

table Monster {
   pos:Vec3;
   mana:short = 150;
   hp:short= 100;
   name:string;
   friendly:bool = false (deprecated, priority: 1)
   inventory:[ubyte];
   color:Color = Blue;
   test:Any;
}

root_type Monster;
```

`Weapon` was defined earlier while `Pickup` is not defined in this example.
Tables
Tables are the main way of defining objects in FlatBuffers and consists of a name (here `Monster`) and a list of fields. Each field has a name, a type, and optionally a default value. If the default value is not specified in the schema, it will be `0` for scalar types, or `null` for other types. Some languages support setting scalar default to `null`. This makes the scalar optional.
Fields do not have to appear in the wire representation and you can choose to omit fields when constructing an object. You have the flexibility of adding fields without bloating your data. This design is also FlatBuffer’s mechanism for forward and backward compatibility. Note that:

- You can add new fields only at the end of a table definition. Older data will still read correctly and give you the default value when read. Older code will simply ignore the new field. If you want to have flexibility to use any order for the fields in your schema, you can manually assign ids (much like Protocol Buffers), see the `id` attribute below.
- You cannot delete fields you do not use anymore from the schema, but you can simply stop writing them into your data for almost the same effect. Additionally you can mark them as deprecated as in the example above which will prevent the generation of accessors in the generated C++, as a way to enforce the field not being used any more (this could break older code).
- You may change field names and table names but do not forget to rename the same in your code as well

Schema evolution examples – these are examples which clarify what happens when you change the schema. We have the following original schema:

```
table { a:int; b:int }
```

and we extend it:

```
table { a:int; b:int; c:int; }
```

This is OK. Code compiled with the old schema reading data generated with the new one will simply ignore the presence of the new field. Code compiled with the new schema reading old data will get the default value for `c` (which is `0` in this case since it is not specified).

```
table { a:int (deprecated); b:int; }
```

This is also ok. Code compiled with the old schema reading new data will now always get the default value for a since it is not present. Code compiled with the old schema reading newer data will now always get the default value for a since it is not present. Code compiled with the new schema now cannot read nor write a anymore (any existing code that tries to do so will result in compile errors), but can still read old data (they will ignore the field).

```
table { c:int; a:int; b:int; }
```

This is NOT ok, as this makes the schemas incompatible. Old code reading newer data will interpret `c` as if it was `a`, and new code reading old data accessing `a` will instead receive `b`.

```
table { c:int (id: 2); a:int (id: 0); b:int (id: 1) }
```

This is ok. If your intent was to order/group fields in a way that makes sense semantically, you can do so using explicit id assignment. Now we are compatible with the original schema and the fields can be ordered in any way, as long as it keeps the sequence of ids.

```
table { b:int; }
```

Not ok. We can remove a field only by deprecation regardless of wether we use explicit ids or not.

```
table { a:uint; b:uint; }
```

This is MAYBE ok, and only in the case where the type change is the same size, like here. If old data never contained any negative numbers this will be safe to do.

```
table { a:int = 1;  b:int = 2; }
```

Generally NOT ok. Any older data written that had `0` values were not written to the buffer, and rely on the default value to be recreated. These will not have those values appear to 1 and 2 instead. There will be cases in which this is OK but care must be taken.

```
table { aa: int; bb: int }
```

Occasionally OK. You have renamed fields which will break all code and JSON files which use this schema, but as long as the change is obvious this is not incompatible with the actual binary since those ever address fields by id/offset.

Structs
Similar to a table only now none of the fields are optional and fields may not be added or deprecated. Structs may only contain scalars or other structs. Use this for simple objects where you are very sure no changes will ever be made (as quite clear in the example `Vec3`). Structs use less memory than tables and are even faster to access (they are always stored inline in their parent object and use no virtual table).

Types
Built-in scalar types are:
8 bit : byte (int8), ubyte (uint8), bool
16 bit:

Apache Arrow

https://github.com/apache/arrow
https://arrow.apache.org/docs/format/Columnar.html

From https://arrow.apache.org/overview/ :

Apache Arrow defines language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware like CPUs and GPUs. The Arrow memory format supports zero copy reads for fast data access without serialization overhead.
Apache Arrow is software development platform for building high-performance applications that process and transport large data sets. It is designed to both improve the performance of analytical algorithms and the efficiency of moving data from one system or one programming language to another. The defining component of Arrow is its in-memory columnar format which is a standardized, language-agnostic specification for representing structured, table-like datasets in memory. This format has rich data type system including nested and user-defined data types designed to support the needs for analytic database systems, data frame libraries, etc.
The Apache Arrow format allows computational routines and execution engines to maximize their efficiency when scanning and iterating large chunks of data. In particular, the contiguous columnar layout enables vectorization using the latest SIMD operations included in modern processors.
Assuming standard format for the data speeds up execution
Without standard format there are potentially a lot of conversion and transforming of data which is lots of unnecessary operations. Moving data from one system to another involves costly serialization and deserialization. In addition, common algorithms must be transformed / rewritten for each data format.

Arrow's in-memory columnar format provides a solution to these kinds of problems allowing data transfers between between disparate systems to be achieved with very low cost. Additionally, the standardized format allows for reuse of libraries and algorithms.

The Arrow Columnar Format
The Arrow Columnar Format includes a language-agnostic in-memory data structure specification, metadata serialization, and a protocol for serialization and generic data transport. The new columnar format is created without the aid of existing implementation. For the implementation of the columnar format Flatbuffers is used for metadata serialization purposes.

Apache Parquet

https://github.com/apache/parquet-cpp
https://parquet.apache.org/documentation/latest/
https://github.com/julienledem/redelm/wiki/The-striping-and-assembly-algorithms-from-the-Dremel-paper
Dremel: interactive Analysis of Web Scale Datasets
//TODO

Swig

https://github.com/swig/swig
http://www.swig.org/tutorial.html
//TODO

Intel MKL

https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html
//TODO

OpenBLAS

https://www.openblas.net/
https://github.com/blas-lapack-rs/openblas-src
//TODO

Atlas

https://github.com/math-atlas/math-atlas
https://sourceforge.net/projects/math-atlas/
https://en.wikipedia.org/wiki/Automatically_Tuned_Linear_Algebra_Software
//TODO

Notes on Pandas design and architecture

//TODO

Understanding Pandas source code

//TODO

Appendix

Python C-API

PyUnicode_Type

```
PyTypeObject PyUnicode_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "str",                          /* tp_name */
    sizeof(PyUnicodeObject),        /* tp_basicsize */
    0,                              /* tp_itemsize */
    /* Slots */
    (destructor)unicode_dealloc,    /* tp_dealloc */
    0,                              /* tp_vectorcall_offset */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_as_async */
    unicode_repr,                   /* tp_repr */
    &unicode_as_number,             /* tp_as_number */
    &unicode_as_sequence,           /* tp_as_sequence */
    &unicode_as_mapping,            /* tp_as_mapping */
    (hashfunc) unicode_hash,        /* tp_hash*/
    0,                              /* tp_call*/
    (reprfunc) unicode_str,         /* tp_str */
    PyObject_GenericGetAttr,        /* tp_getattro */
    0,                              /* tp_setattro */
    0,                              /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
        Py_TPFLAGS_UNICODE_SUBCLASS |
        _Py_TPFLAGS_MATCH_SELF, /* tp_flags */
    unicode_doc,                    /* tp_doc */
    0,                              /* tp_traverse */
    0,                              /* tp_clear */
    PyUnicode_RichCompare,          /* tp_richcompare */
    0,                              /* tp_weaklistoffset */
    unicode_iter,                   /* tp_iter */
    0,                              /* tp_iternext */
    unicode_methods,                /* tp_methods */
    0,                              /* tp_members */
    0,                              /* tp_getset */
    &PyBaseObject_Type,             /* tp_base */
    0,                              /* tp_dict */
    0,                              /* tp_descr_get */
    0,                              /* tp_descr_set */
    0,                              /* tp_dictoffset */
    0,                              /* tp_init */
    0,                              /* tp_alloc */
    unicode_new,                    /* tp_new */
    PyObject_Del,                   /* tp_free */
};
```