UFuncs and DTypes: New possibilities in NumPy

July 15th, SciPy 2022, Austin TX

Sebastian Berg, Stéfan van der Walt

BIDS, UC Berkeley; NumPy

NumPy

BIDS
BERKELEY INSTITUTE FOR DATA SCIENCE

Motivation

New DTypes:

- Physical units (e.g. meter)
- Modernized datetimes
- Strings with encodings
- Arbitrary precision
- …

Also:

- Cleaner NumPy DTypes
- Better ufuncs
- Bug fixes

NumPy DTypes: Example UnitDType


```
from unitdtype import UnitDType

# Array containing meters:
arr = np.zeros(
        3, dtype=UnitDType("m"))
#  ...
```

NumPy DTypes: Example UnitDType


```python
from unitdtype import UnitDType

# Array containing meters:
arr = np.zeros(
        3, dtype=UnitDType("m"))
#  ...
```

```python
class UnitDType(np.dtype):
    itemsize = 8

    def special_methods( ... ):
        pass
```

- Instance (also "descriptor" in C):
  `arr.dtype == UnitDType("m")`
- DType (class/type) defines behavior:
  `type(arr.dtype) is UnitDType`

NumPy DTypes: Example UnitDType

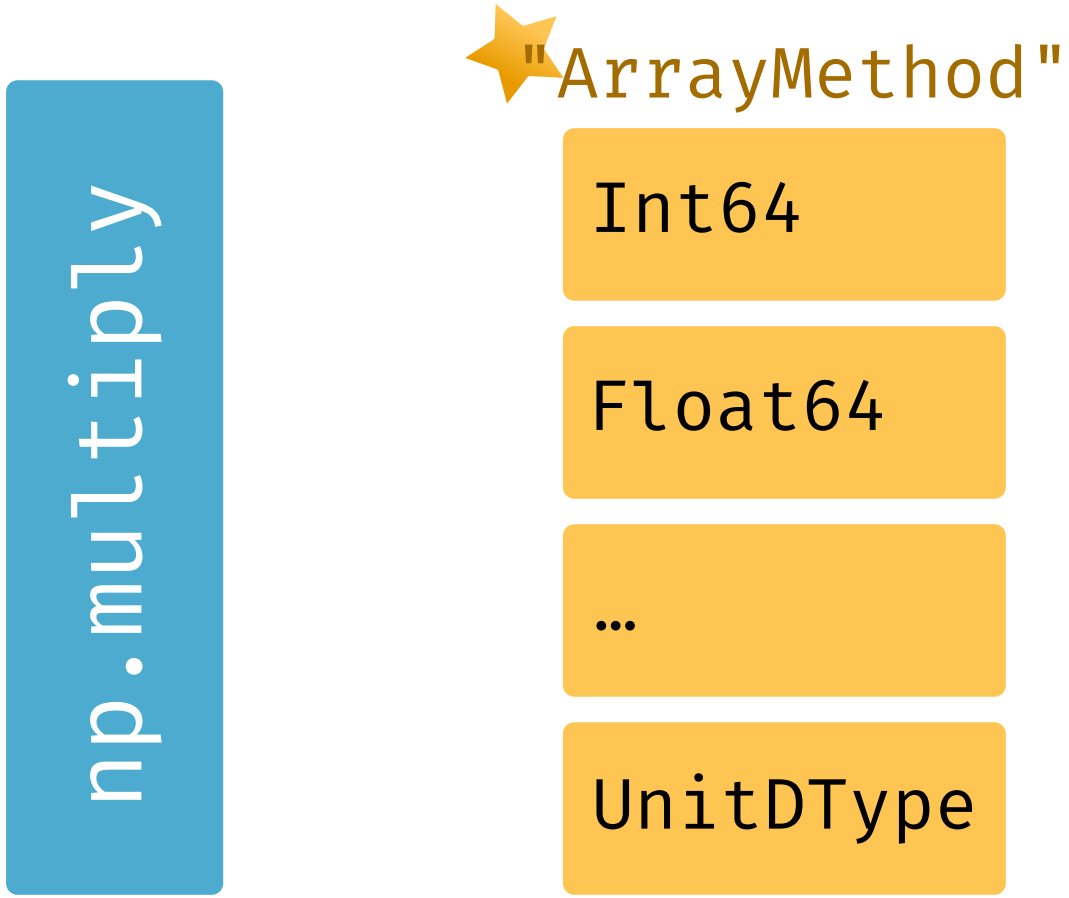
"Experimental" API outlined in NEP 41-43:

```
#include "numpy/experimental_dtype_api.h"

if (import_experimental_dtype_api(5) < 0) {
    return NULL;
}
```

- github.com/seberg/talk_unitdtype (NumPy main branch)
- github.com/seberg/unitdtype (currently own branch)

What is a UFunc?


```
arr1 = np.array([2*m])
arr2 = np.array([3*m])
res = np.multiply(arr1, arr2)
```

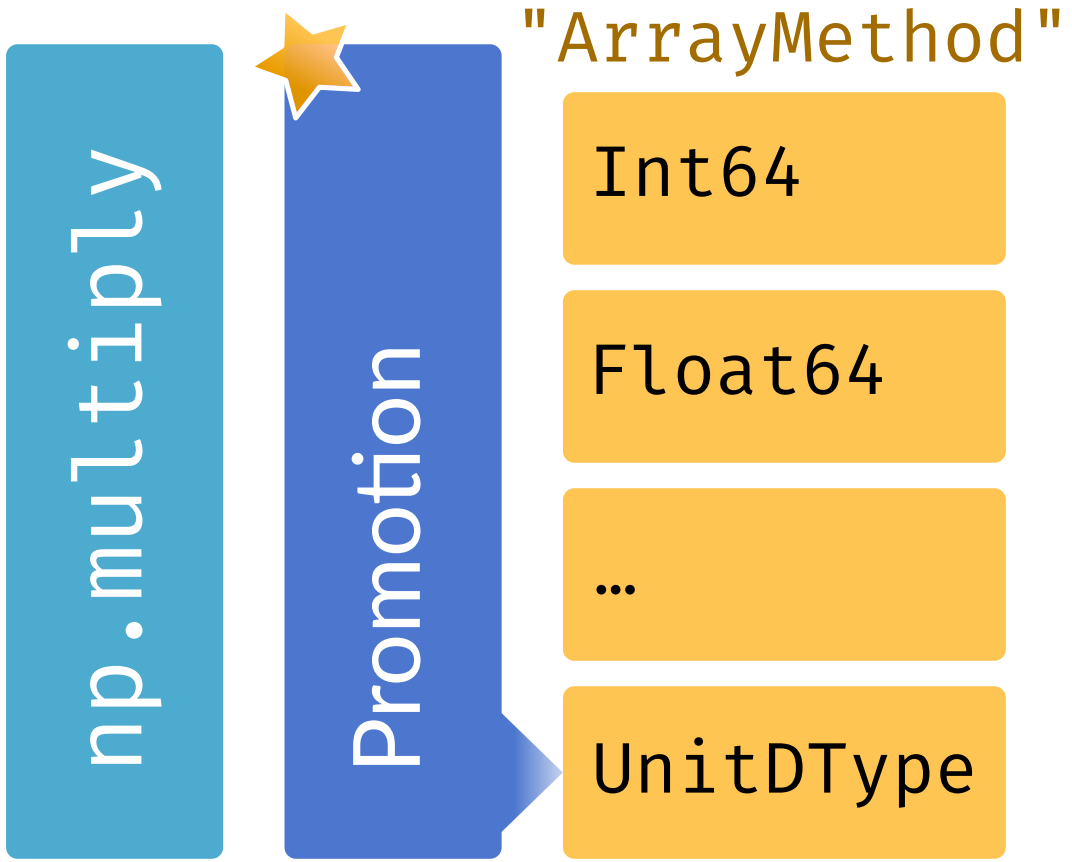


Whats New?

- encapsulation
- extensible

What is a UFunc?


```
arr1 = np.array([2*m])
arr2 = np.array([3*m])
res = np.multiply(arr1, arr2)
```

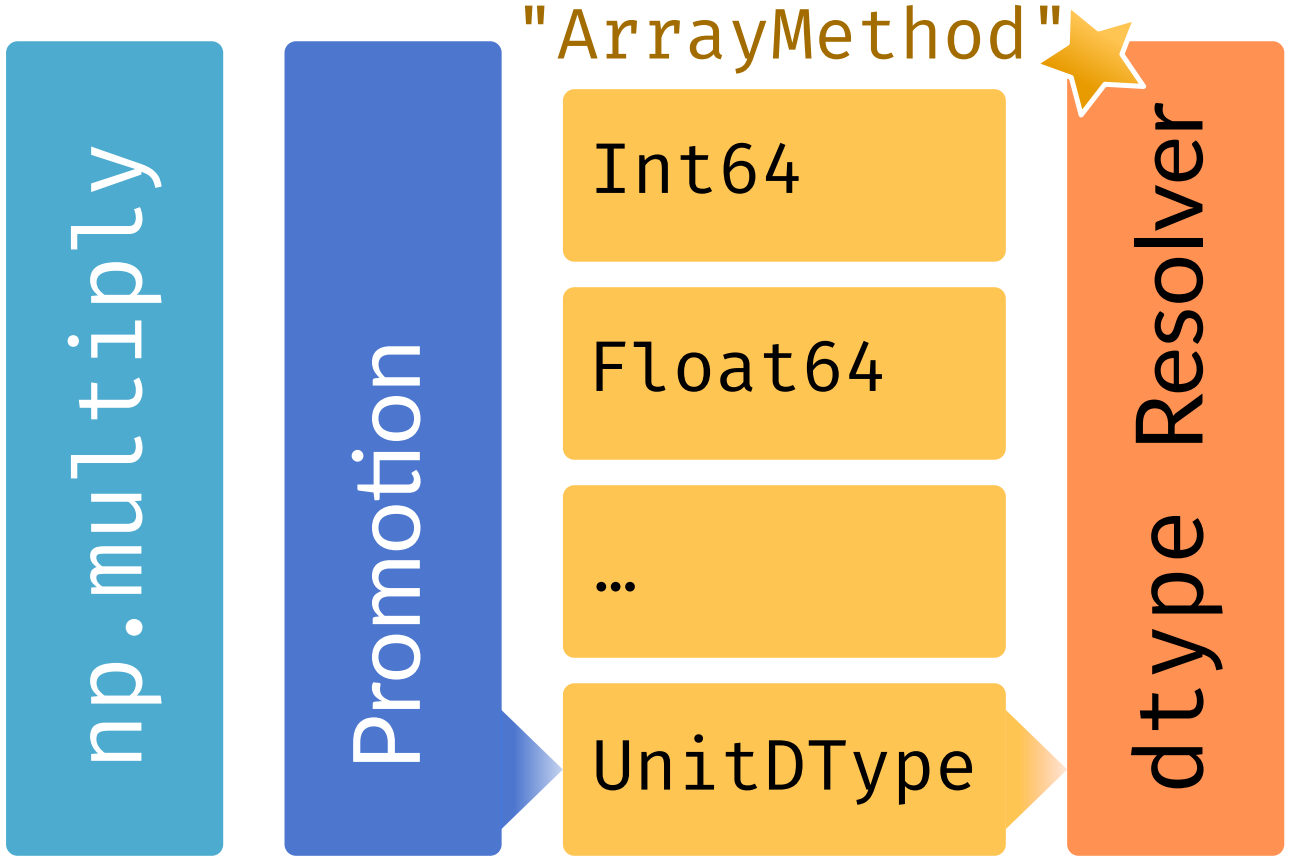


- Whats New?
  - Flexible
  - Customizable
  - useful for numba `DUfunc`

What is a UFunc?


```
arr1 = np.array([2*m])
arr2 = np.array([3*m])
res = np.multiply(arr1, arr2)
```

```
.dtype → "m"
       → "m"
       ← "m**2"
```

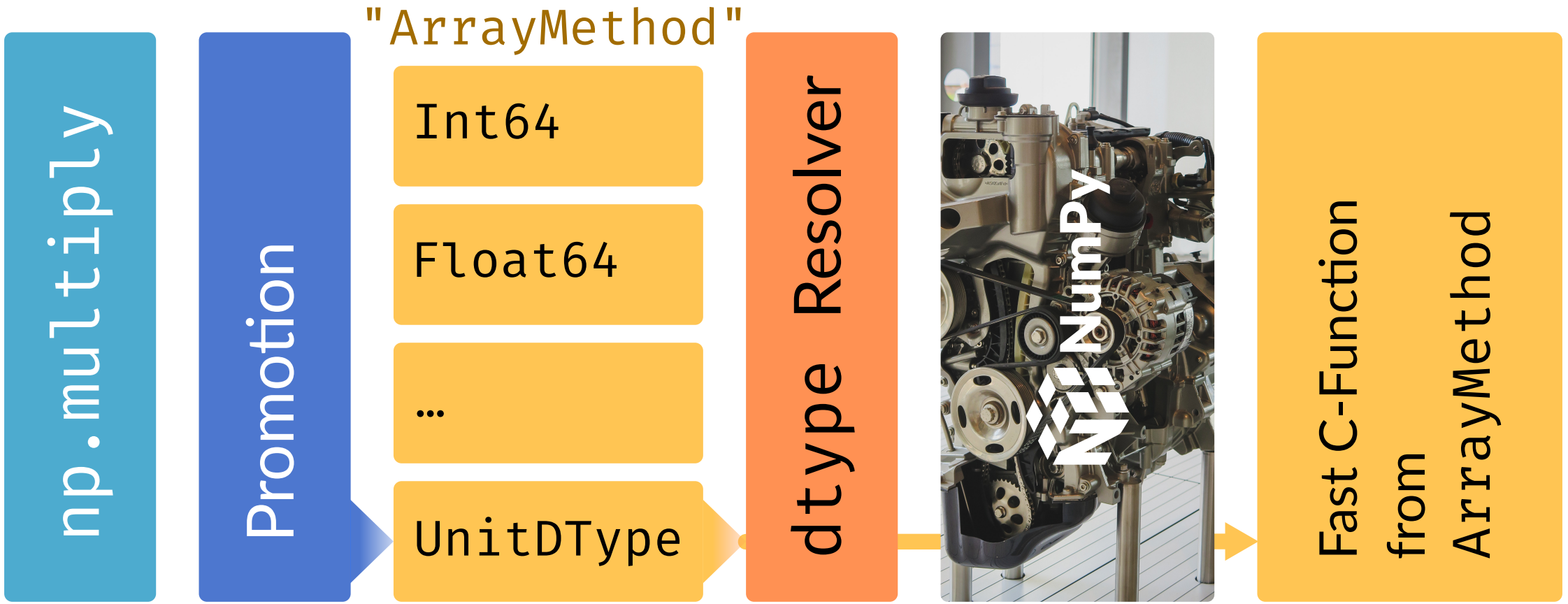


- Whats New?
  - on `ArrayMethod`
  - Customizable
  - …

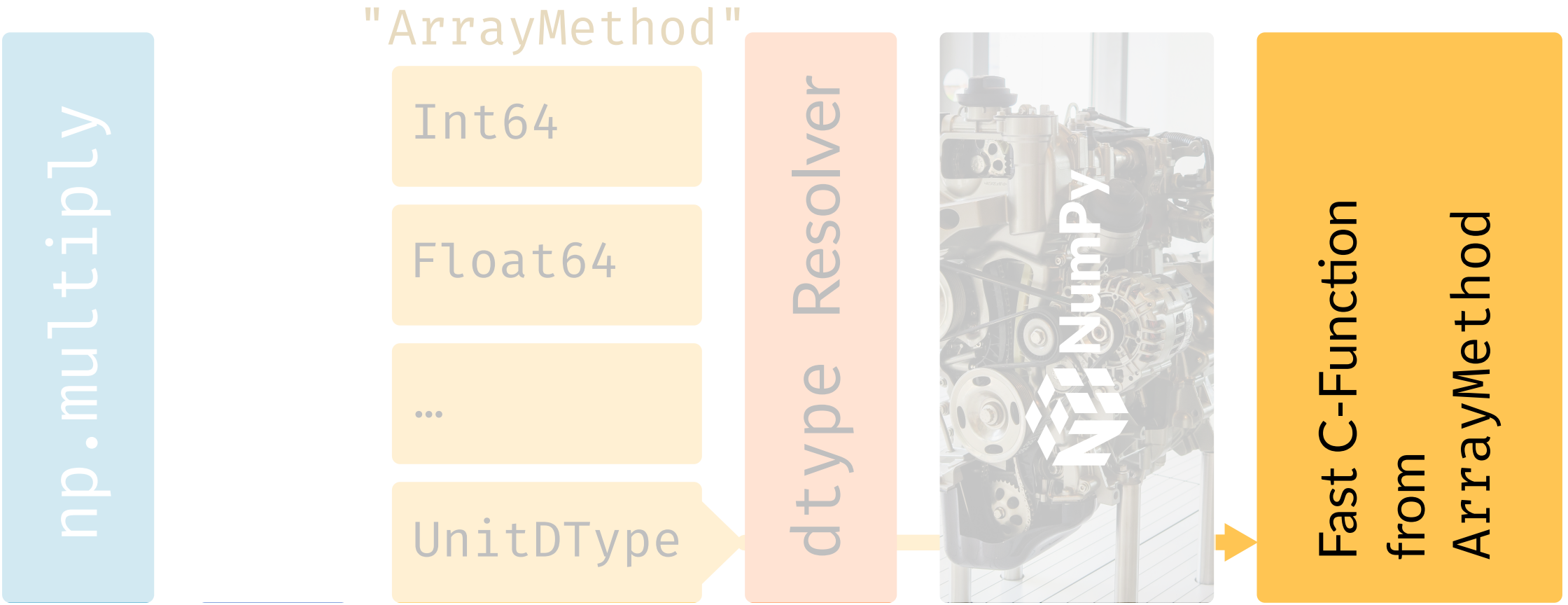
What is a UFunc?


```
arr1 = np.array([2*m])
arr2 = np.array([3*m])
res = np.multiply(arr1, arr2)
```

```
.dtype → "m"
       → "m"
       ← "m**2"
```



Lets make a UFunc: The "strided loop"



Lets make a UFunc: The "strided loop"


```c
static int
unit_multiply_strided_loop(PyArrayMethod_Context *context,
        char *const data[], npy_intp const dimensions[],
        npy_intp const strides[], NpyAuxData *auxdata)
{
    npy_intp N = dimensions[0];
    char *in1 = data[0], *in2 = data[1];
    char *out = data[2];
    npy_intp in1_stride = strides[0];
    npy_intp in2_stride = strides[1];
    npy_intp out_stride = strides[2];

    while (N--) {
        *(double *)out = *(double *)in1 * *(double *)in2;
        in1 += in1_stride;
        in2 += in2_stride;
        out += out_stride;
    }
    return 0;
}
```

Calculate result:

- 1-D loop
- no need for units

```
res[i] = arr1[i] * arr2[i]
```

Lets make a UFunc: The "strided loop"


```c
static int
unit_multiply_strided_loop(PyArrayMethod_Context *context,
        char *const data[], npy_intp const dimensions[],
        npy_intp const strides[], NpyAuxData *auxdata)
{
    npy_intp N = dimensions[0];
    char *in1 = data[0], *in2 = data[1];
    char *out = data[2];
    npy_intp in1_stride = strides[0];
    npy_intp in2_stride = strides[1];
    npy_intp out_stride = strides[2];

    while (N--) {
        *(double *)out = *(double *)in1 * *(double *)in2;
        in1 += in1_stride;
        in2 += in2_stride;
        out += out_stride;
    }
    return 0;
}
```

New "context" with info:

- caller (`np.multiply`)
- "`self`"
- dtypes/descriptors
- ... future additions

Lets make a UFunc: The "strided loop"


```c
static int
unit_multiply_strided_loop(PyArrayMethod_Context *context,
        char *const data[], npy_intp const dimensions[],
        npy_intp const strides[], NpyAuxData *auxdata)
{
    npy_intp N = dimensions[0];
    char *in1 = data[0], *in2 = data[1];
    char *out = data[2];
    npy_intp in1_stride = strides[0];
    npy_intp in2_stride = strides[1];
    npy_intp out_stride = strides[2];

    while (N--) {
        *(double *)out = *(double *)in1 * *(double *)in2;
        in1 += in1_stride;
        in2 += in2_stride;
        out += out_stride;
    }
    return 0;
}
```

Array (current chunks) data:

- `arr1.data`
- `arr2.data`
- `res.data`

Lets make a UFunc: The "strided loop"


```c
static int
unit_multiply_strided_loop(PyArrayMethod_Context *context,
        char *const data[], npy_intp const dimensions[],
        npy_intp const strides[], NpyAuxData *auxdata)
{
    npy_intp N = dimensions[0];
    char *in1 = data[0], *in2 = data[1];
    char *out = data[2];
    npy_intp in1_stride = strides[0];
    npy_intp in2_stride = strides[1];
    npy_intp out_stride = strides[2];

    while (N--) {
        *(double *)out = *(double *)in1 * *(double *)in2;
        in1 += in1_stride;
        in2 += in2_stride;
        out += out_stride;
    }
    return 0;
}
```

The length of the data:

```
N = dimensions[0]
for i in range(N):
    …
```

(more info for generalized UFuncs)

Lets make a UFunc: The "strided loop"


```c
static int
unit_multiply_strided_loop(PyArrayMethod_Context *context,
        char *const data[], npy_intp const dimensions[],
        npy_intp const strides[], NpyAuxData *auxdata)
{
    npy_intp N = dimensions[0];
    char *in1 = data[0], *in2 = data[1];
    char *out = data[2];
    npy_intp in1_stride = strides[0];
    npy_intp in2_stride = strides[1];
    npy_intp out_stride = strides[2];

    while (N--) {
        *(double *)out = *(double *)in1 * *(double *)in2;
        in1 += in1_stride;
        in2 += in2_stride;
        out += out_stride;
    }
    return 0;
}
```

The "`arr.strides[-1]`":

- how to iterate the data

Lets make a UFunc: The "strided loop"


```c
static int
unit_multiply_strided_loop(PyArrayMethod_Context *context,
        char *const data[], npy_intp const dimensions[],
        npy_intp const strides[], NpyAuxData *auxdata)
{
    npy_intp N = dimensions[0];
    char *in1 = data[0], *in2 = data[1];
    char *out = data[2];
    npy_intp in1_stride = strides[0];
    npy_intp in2_stride = strides[1];
    npy_intp out_stride = strides[2];

    while (N--) {
        *(double *)out = *(double *)in1 * *(double *)in2;
        in1 += in1_stride;
        in2 += in2_stride;
        out += out_stride;
    }
    return 0;
}
```

Return:

- success: `0`
- error: `-1`

Important API addition!

Lets make a UFunc: The "strided loop"


```c
static int
unit_multiply_strided_loop(PyArrayMethod_Context *context,
        char *const data[], npy_intp const dimensions[],
        npy_intp const strides[], NpyAuxData *auxdata)
{
    npy_intp N = dimensions[0];
    char *in1 = data[0], *in2 = data[1];
    char *out = data[2];
    npy_intp in1_stride = strides[0];
    npy_intp in2_stride = strides[1];
    npy_intp out_stride = strides[2];

    while (N--) {
        *(double *)out = *(double *)in1 * *(double *)in2;
        in1 += in1_stride;
        in2 += in2_stride;
        out += out_stride;
    }
    return 0;
}
```

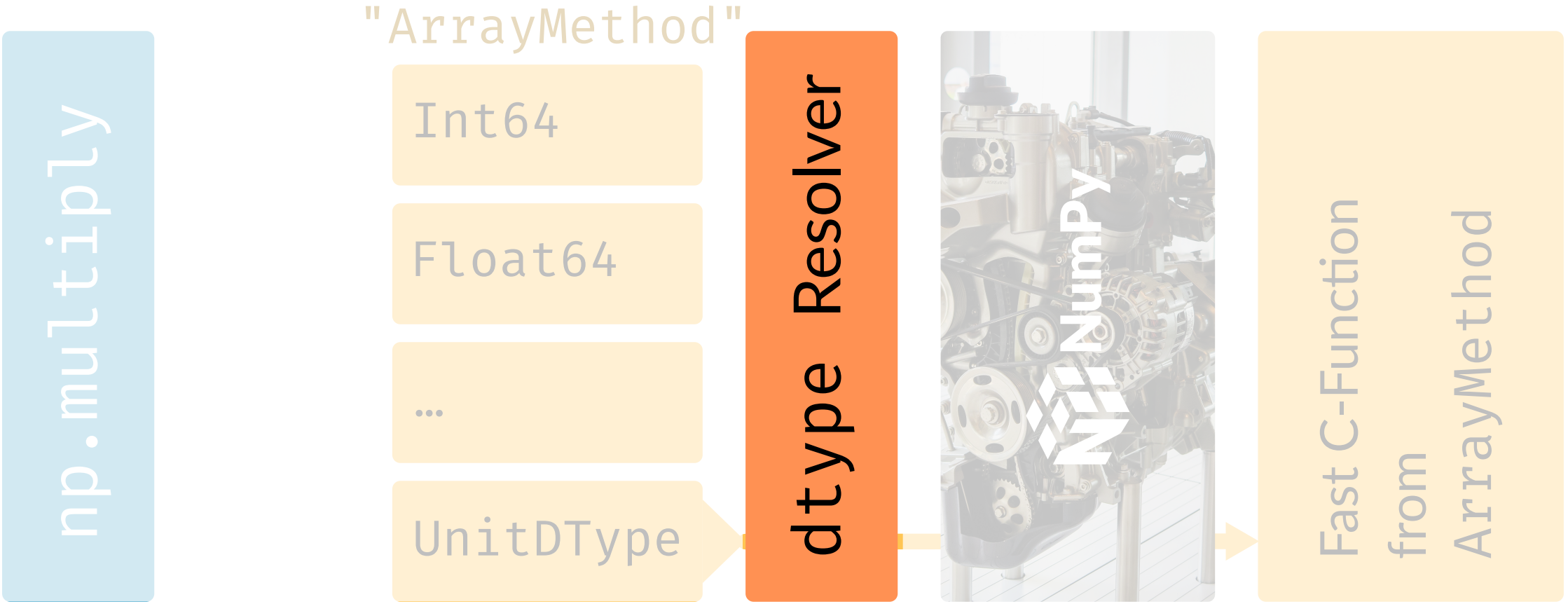
This is an Example!

Loop re-use possible!

`auxdata`:

- pass information
- scratch space

Lets make a UFunc: The "Resolver"



Lets make a UFunc: The "Resolver"


```c
static NPY_CASTING
unit_multiply_resolve_descriptors(PyObject *self,
        PyArray_DTypeMeta *dtypes[], PyArray_Descr *given_descrs[],
        PyArray_Descr *loop_descrs[], npy_intp *unused)
{
    /* Fetch the unyt based units: */
    PyObject *unit1 = ((UnitDTypeObject *)given_descrs[0])->unit;
    PyObject *unit2 = ((UnitDTypeObject *)given_descrs[1])->unit;
    /* Find the correct result unit: */
    PyObject *new_unit = PyNumber_Multiply(unit1, unit2);
    if (new_unit == NULL) {
        return -1;
    }
    /* Create new DType from the new unit: */
    loop_descrs[2] = new_unitdtype_instance(new_unit);
    if (loop_descrs[2] == NULL) {
        return -1;
    }
    /* The other operand units can be used as-is: */
    Py_INCREF(given_descrs[0]);
    loop_descrs[0] = given_descrs[0];
    Py_INCREF(given_descrs[1]);
    loop_descrs[1] = given_descrs[0];

    return NPY_NO_CASTING;
}
```

User code:

```
arr1 = np.array([2*m])
arr2 = np.array([3*m])
res = arr1 * arr2
```

Resolver finds:

```
dtype1 = arr1.dtype  # "m"
dtype2 = arr2.dtype  # "m"
res_dt = res.dtype   # "m**2"
```

Lets make a UFunc: The "Resolver"


```c
static NPY_CASTING
unit_multiply_resolve_descriptors(PyObject *self,
        PyArray_DTypeMeta *dtypes[], PyArray_Descr *given_descrs[],
        PyArray_Descr *loop_descrs[], npy_intp *unused)
{
    /* Fetch the unyt based units: */
    PyObject *unit1 = ((UnitDTypeObject *)given_descrs[0])->unit;
    PyObject *unit2 = ((UnitDTypeObject *)given_descrs[1])->unit;
    /* Find the correct result unit: */
    PyObject *new_unit = PyNumber_Multiply(unit1, unit2);
    if (new_unit == NULL) {
        return -1;
    }
    /* Create new DType from the new unit: */
    loop_descrs[2] = new_unitdtype_instance(new_unit);
    if (loop_descrs[2] == NULL) {
        return -1;
    }
    /* The other operand units can be used as-is: */
    Py_INCREF(given_descrs[0]);
    loop_descrs[0] = given_descrs[0];
    Py_INCREF(given_descrs[1]);
    loop_descrs[1] = given_descrs[0];

    return NPY_NO_CASTING;
}
```

"self":

- fairly opaque currently
- Future information

Lets make a UFunc: The "Resolver"


```c
static NPY_CASTING
unit_multiply_resolve_descriptors(PyObject *self,
        PyArray_DTypeMeta *dtypes[], PyArray_Descr *given_descrs[],
        PyArray_Descr *loop_descrs[], npy_intp *unused)
{
    /* Fetch the unyt based units: */
    PyObject *unit1 = ((UnitDTypeObject *)given_descrs[0])->unit;
    PyObject *unit2 = ((UnitDTypeObject *)given_descrs[1])->unit;
    /* Find the correct result unit: */
    PyObject *new_unit = PyNumber_Multiply(unit1, unit2);
    if (new_unit == NULL) {
        return -1;
    }
    /* Create new DType from the new unit: */
    loop_descrs[2] = new_unitdtype_instance(new_unit);
    if (loop_descrs[2] == NULL) {
        return -1;
    }
    /* The other operand units can be used as-is: */
    Py_INCREF(given_descrs[0]);
    loop_descrs[0] = given_descrs[0];
    Py_INCREF(given_descrs[1]);
    loop_descrs[1] = given_descrs[0];

    return NPY_NO_CASTING;
}
```

The DTypes:

- just: `UnitDType`
- allows generic resolvers

Lets make a UFunc: The "Resolver"


```c
static NPY_CASTING
unit_multiply_resolve_descriptors(PyObject *self,
        PyArray_DTypeMeta *dtypes[], PyArray_Descr *given_descrs[],
        PyArray_Descr *loop_descrs[], npy_intp *unused)
{
    /* Fetch the unyt based units: */
    PyObject *unit1 = ((UnitDTypeObject *)given_descrs[0])->unit;
    PyObject *unit2 = ((UnitDTypeObject *)given_descrs[1])->unit;
    /* Find the correct result unit: */
    PyObject *new_unit = PyNumber_Multiply(unit1, unit2);
    if (new_unit == NULL) {
        return -1;
    }
    /* Create new DType from the new unit: */
    loop_descrs[2] = new_unitdtype_instance(new_unit);
    if (loop_descrs[2] == NULL) {
        return -1;
    }
    /* The other operand units can be used as-is: */
    Py_INCREF(given_descrs[0]);
    loop_descrs[0] = given_descrs[0];
    Py_INCREF(given_descrs[1]);
    loop_descrs[1] = given_descrs[0];

    return NPY_NO_CASTING;
}
```

The dtypes / descriptors:

- `arr1.dtype`
- `arr2.dtype`
- `NULL` (or `out.dtype`)

Lets make a UFunc: The "Resolver"


```c
static NPY_CASTING
unit_multiply_resolve_descriptors(PyObject *self,
        PyArray_DTypeMeta *dtypes[], PyArray_Descr *given_descrs[],
        PyArray_Descr *loop_descrs[], npy_intp *unused)
{
    /* Fetch the unyt based units: */
    PyObject *unit1 = ((UnitDTypeObject *)given_descrs[0])->unit;
    PyObject *unit2 = ((UnitDTypeObject *)given_descrs[1])->unit;
    /* Find the correct result unit: */
    PyObject *new_unit = PyNumber_Multiply(unit1, unit2);
    if (new_unit == NULL) {
        return -1;
    }
    /* Create new DType from the new unit: */
    loop_descrs[2] = new_unitdtype_instance(new_unit);
    if (loop_descrs[2] == NULL) {
        return -1;
    }
    /* The other operand units can be used as-is: */
    Py_INCREF(given_descrs[0]);
    loop_descrs[0] = given_descrs[0];
    Py_INCREF(given_descrs[1]);
    loop_descrs[1] = given_descrs[0];

    return NPY_NO_CASTING;
}
```

Output the dtypes/descriptors to use in the C-loop:

- ?
- ?
- ?

Lets make a UFunc: The "Resolver"


```c
static NPY_CASTING
unit_multiply_resolve_descriptors(PyObject *self,
        PyArray_DTypeMeta *dtypes[], PyArray_Descr *given_descrs[],
        PyArray_Descr *loop_descrs[], npy_intp *unused)
{
    /* Fetch the unyt based units: */
    PyObject *unit1 = ((UnitDTypeObject *)given_descrs[0])->unit;
    PyObject *unit2 = ((UnitDTypeObject *)given_descrs[1])->unit;
    /* Find the correct result unit: */
    PyObject *new_unit = PyNumber_Multiply(unit1, unit2);
    if (new_unit == NULL) {
        return -1;
    }
    /* Create new DType from the new unit: */
    loop_descrs[2] = new_unitdtype_instance(new_unit);
    if (loop_descrs[2] == NULL) {
        return -1;
    }
    /* The other operand units can be used as-is: */
    Py_INCREF(given_descrs[0]);
    loop_descrs[0] = given_descrs[0];
    Py_INCREF(given_descrs[1]);
    loop_descrs[1] = given_descrs[0];

    return NPY_NO_CASTING;
}
```

Find the result unit:

```
unit1 = arr1.dtype.unit
unit2 = arr2.dtype.unit

new_unit = unit1 * unit2
```

Lets make a UFunc: The "Resolver"


```c
static NPY_CASTING
unit_multiply_resolve_descriptors(PyObject *self,
        PyArray_DTypeMeta *dtypes[], PyArray_Descr *given_descrs[],
        PyArray_Descr *loop_descrs[], npy_intp *unused)
{
    /* Fetch the unyt based units: */
    PyObject *unit1 = ((UnitDTypeObject *)given_descrs[0])->unit;
    PyObject *unit2 = ((UnitDTypeObject *)given_descrs[1])->unit;
    /* Find the correct result unit: */
    PyObject *new_unit = PyNumber_Multiply(unit1, unit2);
    if (new_unit == NULL) {
        return -1;
    }
    /* Create new DType from the new unit: */
    loop_descrs[2] = new_unitdtype_instance(new_unit);
    if (loop_descrs[2] == NULL) {
        return -1;
    }
    /* The other operand units can be used as-is: */
    Py_INCREF(given_descrs[0]);
    loop_descrs[0] = given_descrs[0];
    Py_INCREF(given_descrs[1]);
    loop_descrs[1] = given_descrs[0];

    return NPY_NO_CASTING;
}
```

Result dtype/descriptor:

```
res_dt = UnitDType(new_unit)
```

Lets make a UFunc: The "Resolver"


```c
static NPY_CASTING
unit_multiply_resolve_descriptors(PyObject *self,
        PyArray_DTypeMeta *dtypes[], PyArray_Descr *given_descrs[],
        PyArray_Descr *loop_descrs[], npy_intp *unused)
{
    /* Fetch the unyt based units: */
    PyObject *unit1 = ((UnitDTypeObject *)given_descrs[0])->unit;
    PyObject *unit2 = ((UnitDTypeObject *)given_descrs[1])->unit;
    /* Find the correct result unit: */
    PyObject *new_unit = PyNumber_Multiply(unit1, unit2);
    if (new_unit == NULL) {
        return -1;
    }
    /* Create new DType from the new unit: */
    loop_descrs[2] = new_unitdtype_instance(new_unit);
    if (loop_descrs[2] == NULL) {
        return -1;
    }
    /* The other operand units can be used as-is: */
    Py_INCREF(given_descrs[0]);
    loop_descrs[0] = given_descrs[0];
    Py_INCREF(given_descrs[1]);
    loop_descrs[1] = given_descrs[0];

    return NPY_NO_CASTING;
}
```

Output the dtypes/descriptors to use in the C-loop:

- `arr1.dtype`
- `arr2.dtype`
- `res_dt`

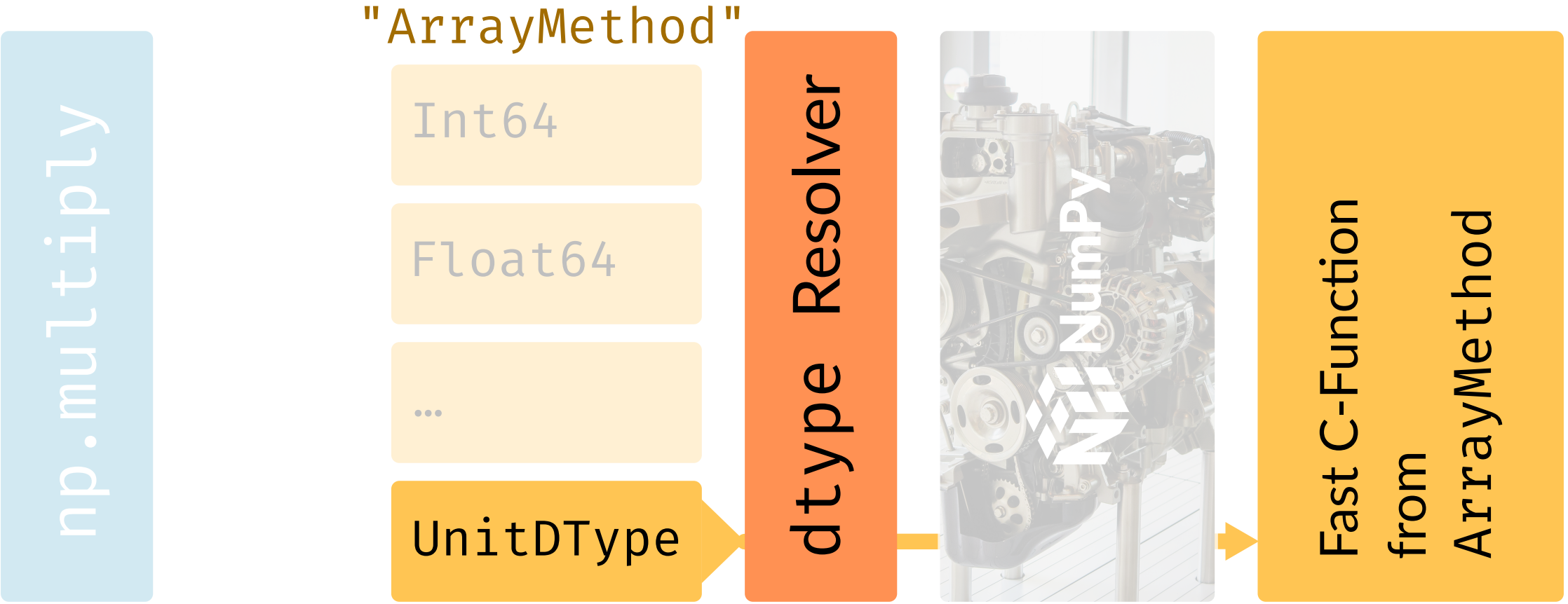
Lets make a UFunc: The "Resolver"


```c
static NPY_CASTING
unit_multiply_resolve_descriptors(PyObject *self,
        PyArray_DTypeMeta *dtypes[], PyArray_Descr *given_descrs[],
        PyArray_Descr *loop_descrs[], npy_intp *unused)
{
    /* Fetch the unyt based units: */
    PyObject *unit1 = ((UnitDTypeObject *)given_descrs[0])->unit;
    PyObject *unit2 = ((UnitDTypeObject *)given_descrs[1])->unit;
    /* Find the correct result unit: */
    PyObject *new_unit = PyNumber_Multiply(unit1, unit2);
    if (new_unit == NULL) {
        return -1;
    }
    /* Create new DType from the new unit: */
    loop_descrs[2] = new_unitdtype_instance(new_unit);
    if (loop_descrs[2] == NULL) {
        return -1;
    }
    /* The other operand units can be used as-is: */
    Py_INCREF(given_descrs[0]);
    loop_descrs[0] = given_descrs[0];
    Py_INCREF(given_descrs[1]);
    loop_descrs[1] = given_descrs[0];

    return NPY_NO_CASTING;
}
```

This UFunc does not cast!

- potential optimizations
- important for casts!

Lets make a UFunc: 🎁 Wrap it up!



Lets make a UFunc: 🎁 Wrap it up!


```
static PyArray_DTypeMeta *dtypes[3] = {
   &UnitDType, &UnitDType, &UnitDType};

static PyType_Slot slots[] = {
   {NPY_METH_resolve_descriptors,
        &unit_multiply_resolve_descriptors},
   {NPY_METH_strided_loop,
        &unit_multiply_strided_loop},
   {0, NULL}
};

PyArrayMethod_Spec MultiplySpec = {
    .name = "unit_multiply",
    .nin = 2,
    .nout = 1,
    .dtypes = dtypes,
    .slots = slots,
    .flags = 0,
    .casting = NPY_NO_CASTING,
};

/* Register */
PyUFunc_AddLoopFromSpec(multiply, &MultiplySpec);
```

Loop operates on `UnitDType`

Lets make a UFunc: 🎁 Wrap it up!


```
static PyArray_DTypeMeta *dtypes[3] = {
   &UnitDType, &UnitDType, &UnitDType};

static PyType_Slot slots[] = {
   {NPY_METH_resolve_descriptors,
        &unit_multiply_resolve_descriptors},
   {NPY_METH_strided_loop,
        &unit_multiply_strided_loop},
   {0, NULL}
};

PyArrayMethod_Spec MultiplySpec = {
    .name = "unit_multiply",
    .nin = 2,
    .nout = 1,
    .dtypes = dtypes,
    .slots = slots,
    .flags = 0,
    .casting = NPY_NO_CASTING,
};

/* Register */
PyUFunc_AddLoopFromSpec(multiply, &MultiplySpec);
```

List of "methods":

- Fill in what we defined
- Extensible

More options (not shown):

- Faster loops
- Set-up e.g. scratch space

Lets make a UFunc: 🎁 Wrap it up!


```c
static PyArray_DTypeMeta *dtypes[3] = {
   &UnitDType, &UnitDType, &UnitDType};

static PyType_Slot slots[] = {
   {NPY_METH_resolve_descriptors,
        &unit_multiply_resolve_descriptors},
   {NPY_METH_strided_loop,
        &unit_multiply_strided_loop},
   {0, NULL}
};

PyArrayMethod_Spec MultiplySpec = {
    .name = "unit_multiply",
    .nin = 2,
    .nout = 1,
    .dtypes = dtypes,
    .slots = slots,
    .flags = 0,
    .casting = NPY_NO_CASTING,
};

/* Register */
PyUFunc_AddLoopFromSpec(multiply, &MultiplySpec);
```

The "Spec" (C `struct`)

- name
- number of inputs/outputs
- DTypes (`UnitDType`)
- `slots`: "methods"
- ...

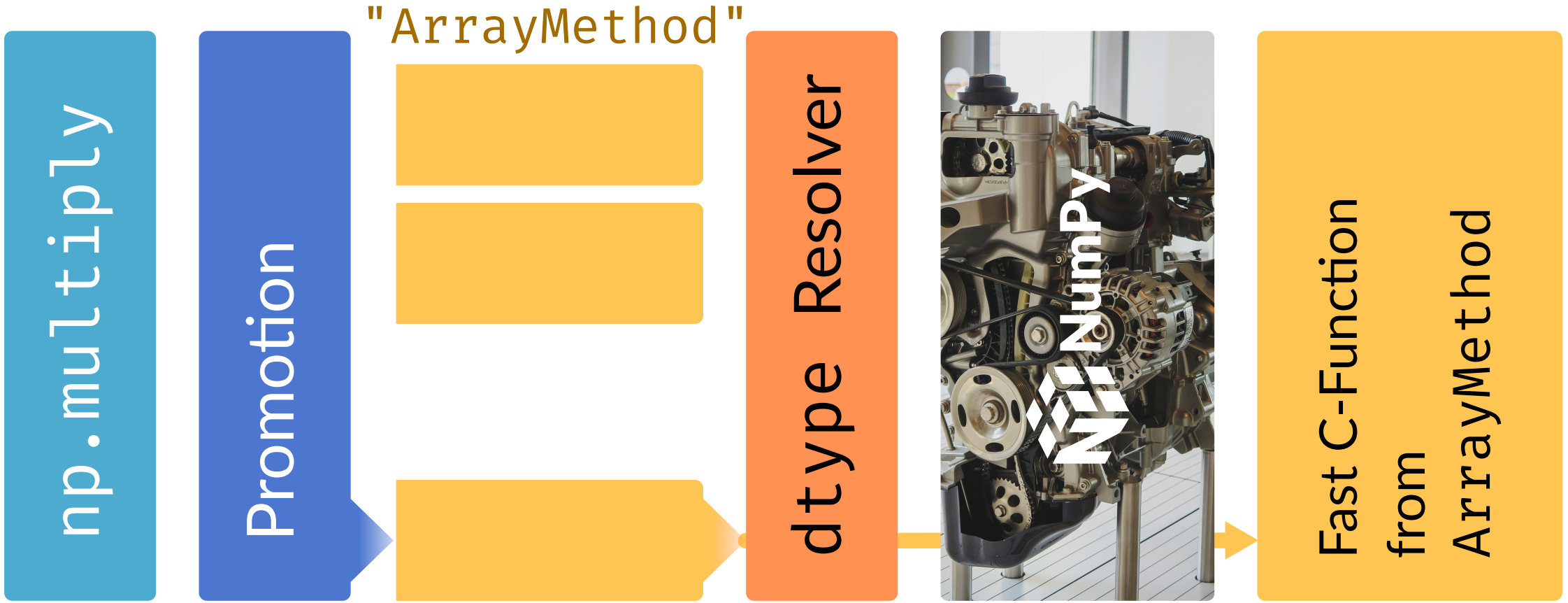
Lets make a UFunc: 🎁 Wrap it up!


```
static PyArray_DTypeMeta *dtypes[3] = {
   &UnitDType, &UnitDType, &UnitDType};

static PyType_Slot slots[] = {
   {NPY_METH_resolve_descriptors,
        &unit_multiply_resolve_descriptors},
   {NPY_METH_strided_loop,
        &unit_multiply_strided_loop},
   {0, NULL}
};

PyArrayMethod_Spec MultiplySpec = {
    .name = "unit_multiply",
    .nin = 2,
    .nout = 1,
    .dtypes = dtypes,
    .slots = slots,
    .flags = 0,
    .casting = NPY_NO_CASTING,
};

/* Register */
PyUFunc_AddLoopFromSpec(multiply, &MultiplySpec);
```

Add to multiply ufunc!

Done with UFunc!


```
arr1 = np.array([2*m])
arr2 = np.array([3*m])
res = np.multiply(arr1, arr2)
```



How do Casts work?


```
arr1 = np.array([2*m])

res = arr1.astype(UnitDType("cm"))
```

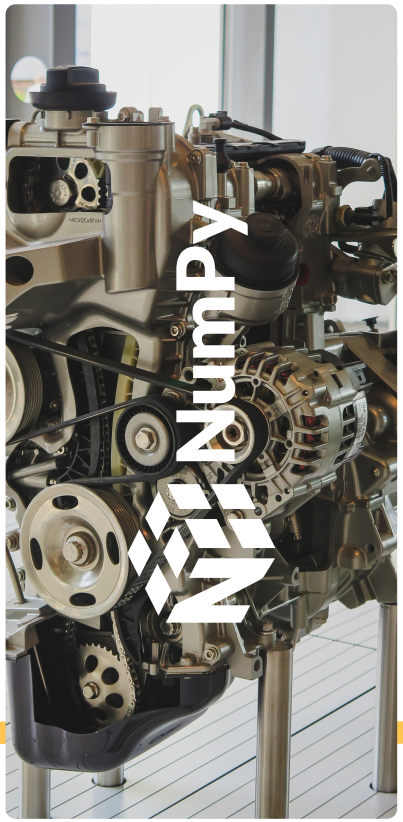
"ArrayMethod"

… → UnitDType

UnitDType → …

UnitDType → UnitDType

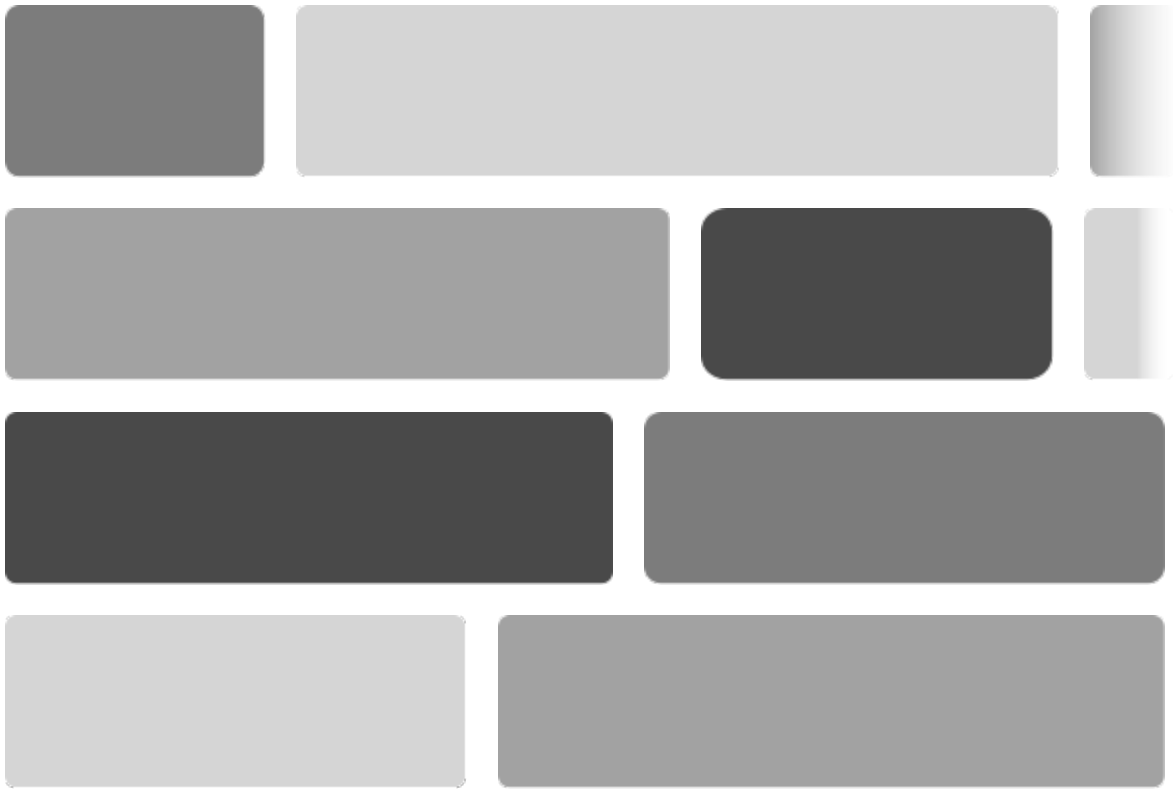
dtype Resolver



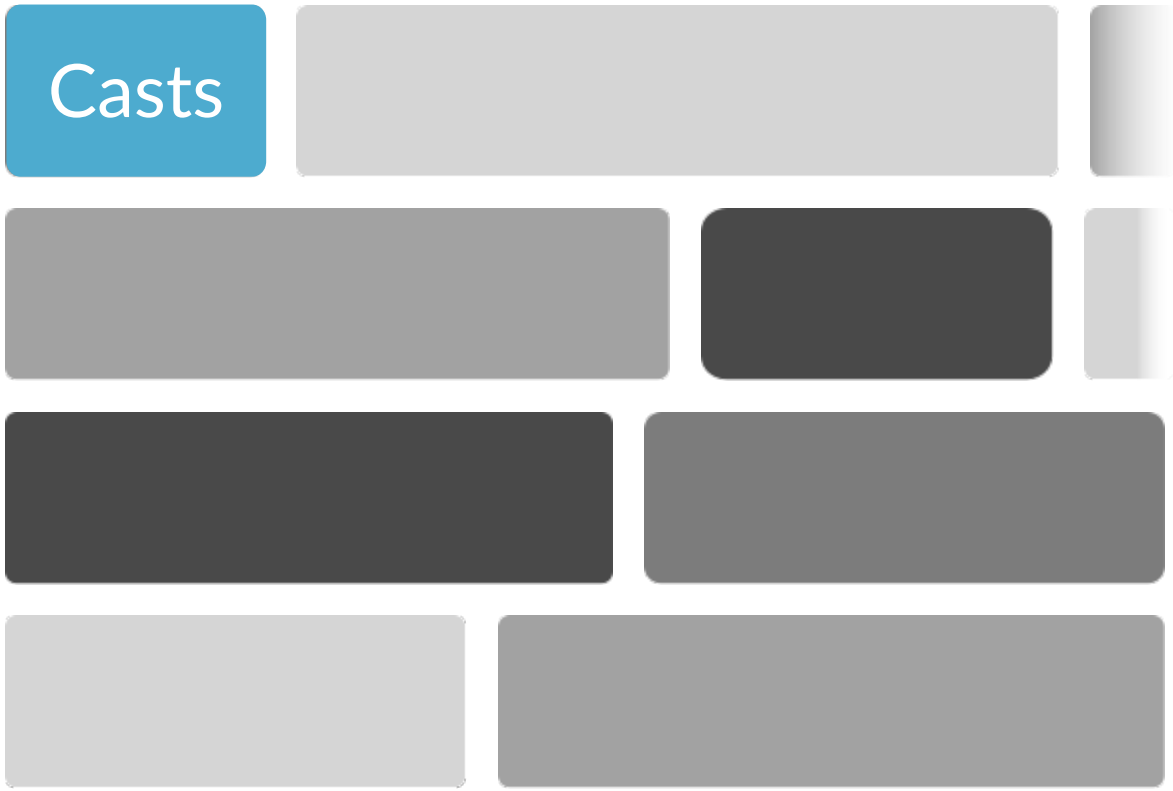
Fast C-Function from ArrayMethod

What makes the UnitDType?


```
class UnitDType(np.dtype):
    itemsize = 8

    def special_methods( ... ):
        pass
```



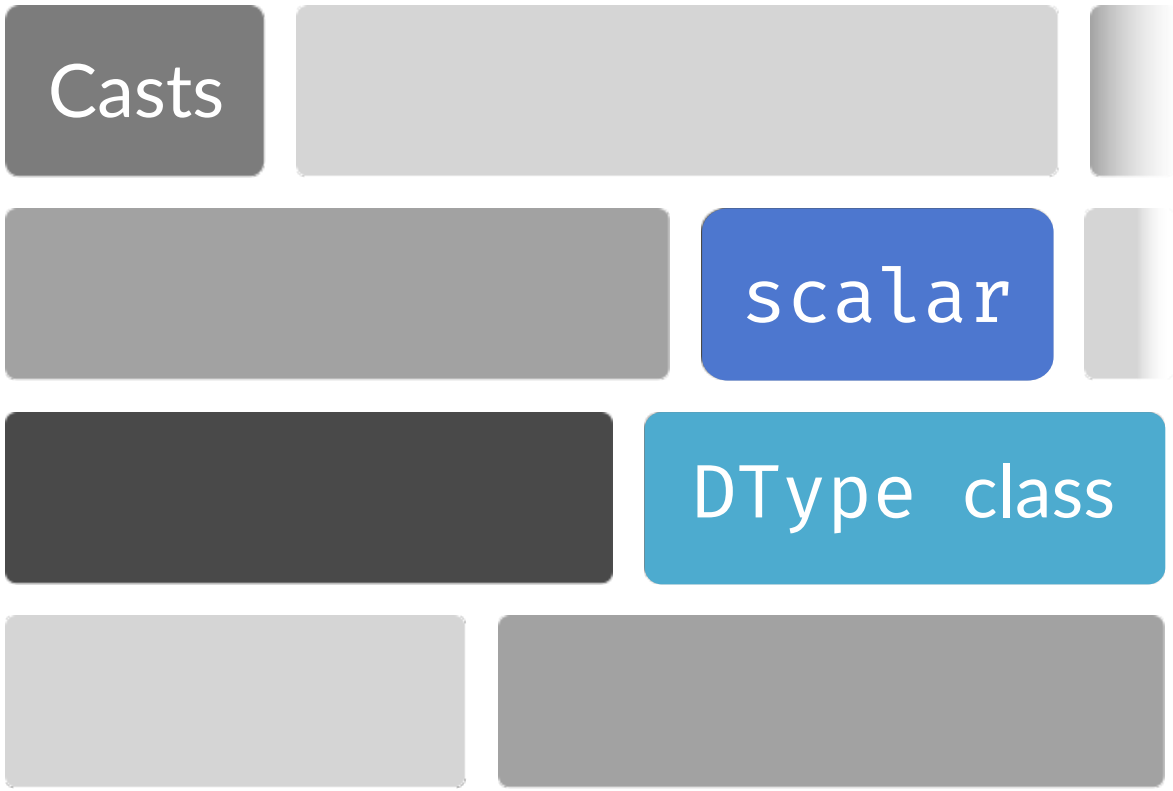
What makes the UnitDType?



What makes the UnitDType?

Type(s):

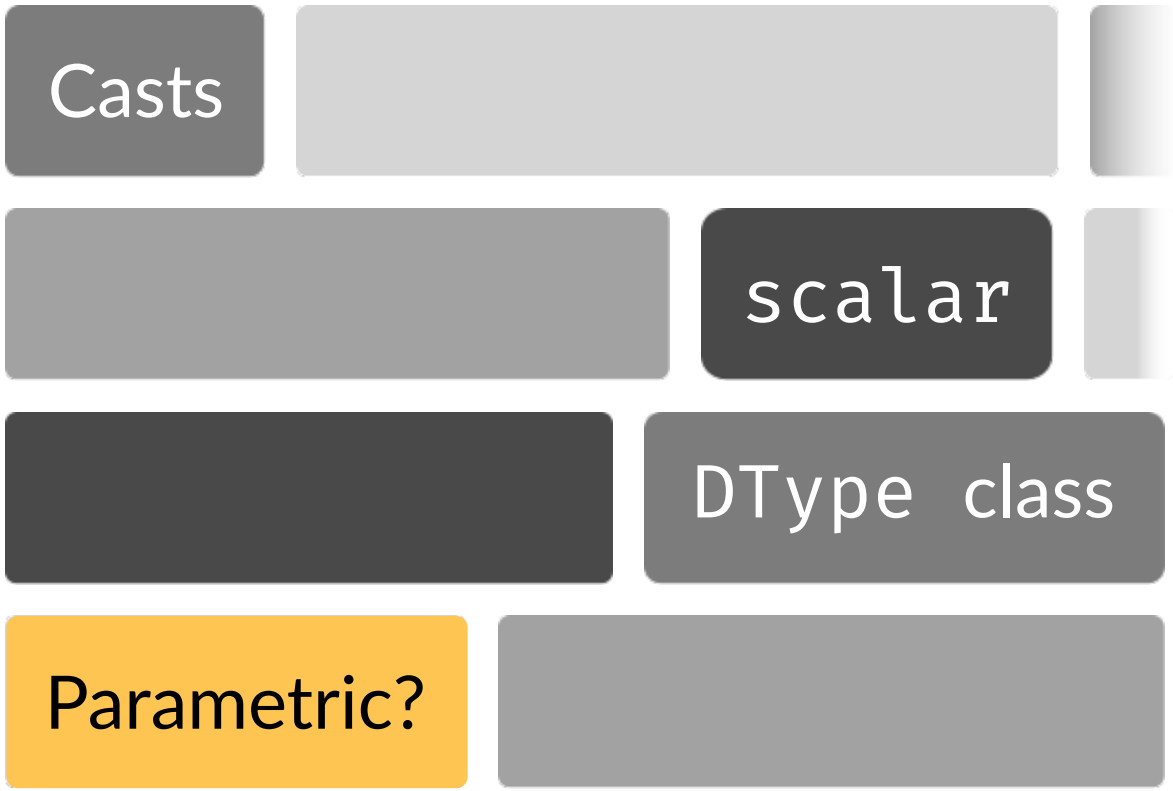
- Scalar Type (`2*m`)
- `UnitDType` class
- Future: no scalar?



What makes the UnitDType?

Flags:

- currently only one



What makes the UnitDType?

Set/Get element:

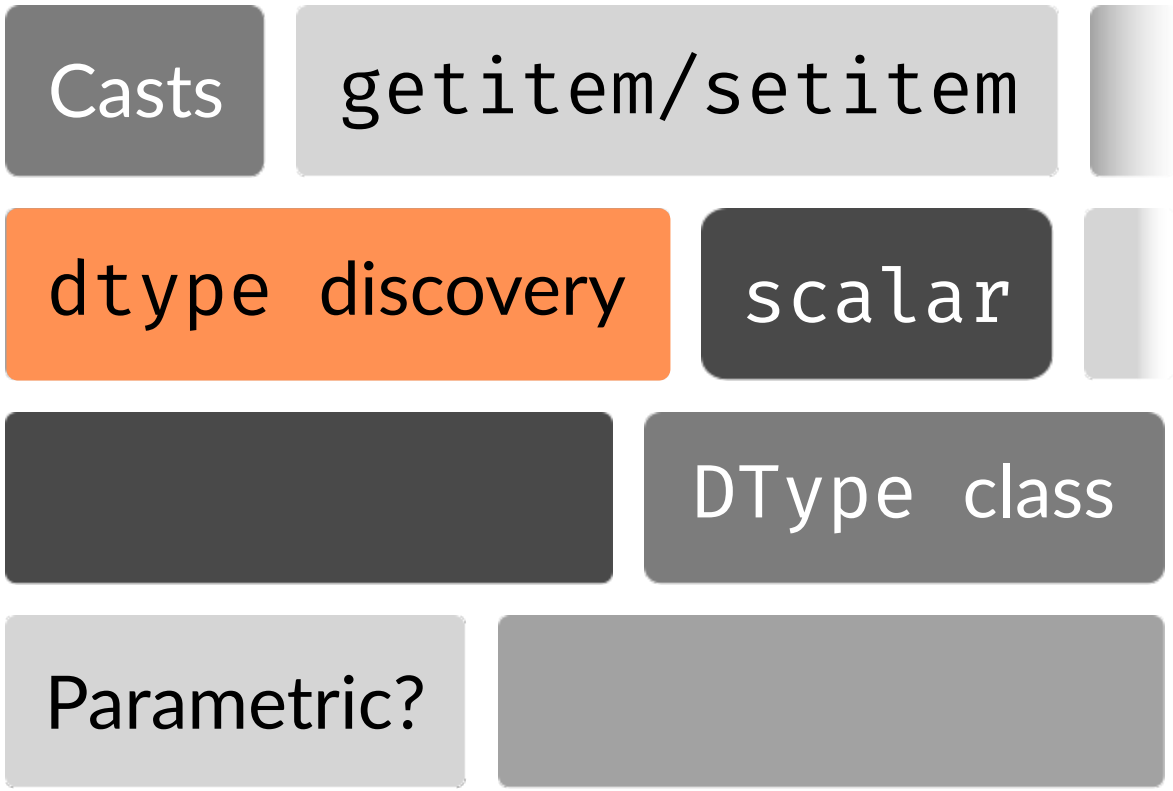
```
arr[i] = 2*m
scalar = arr[i]
```

What makes the UnitDType?

Discover dtype from scalar:

- only if parametric

```
scalar = 2*m
np.array([scalar])
```

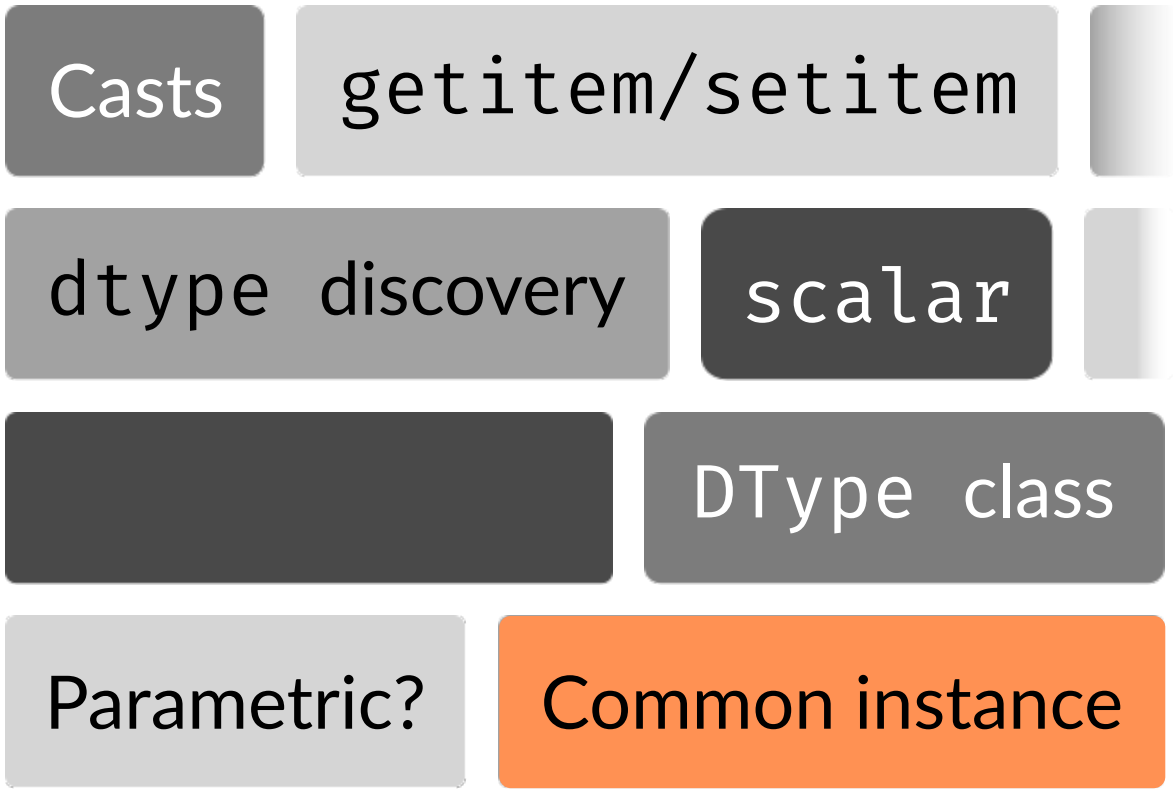


What makes the UnitDType?

Two `UnitDTypes`:

- Find "common" one
- only if parametric

```
np.array([1*m, 2*cm])
  → m
```

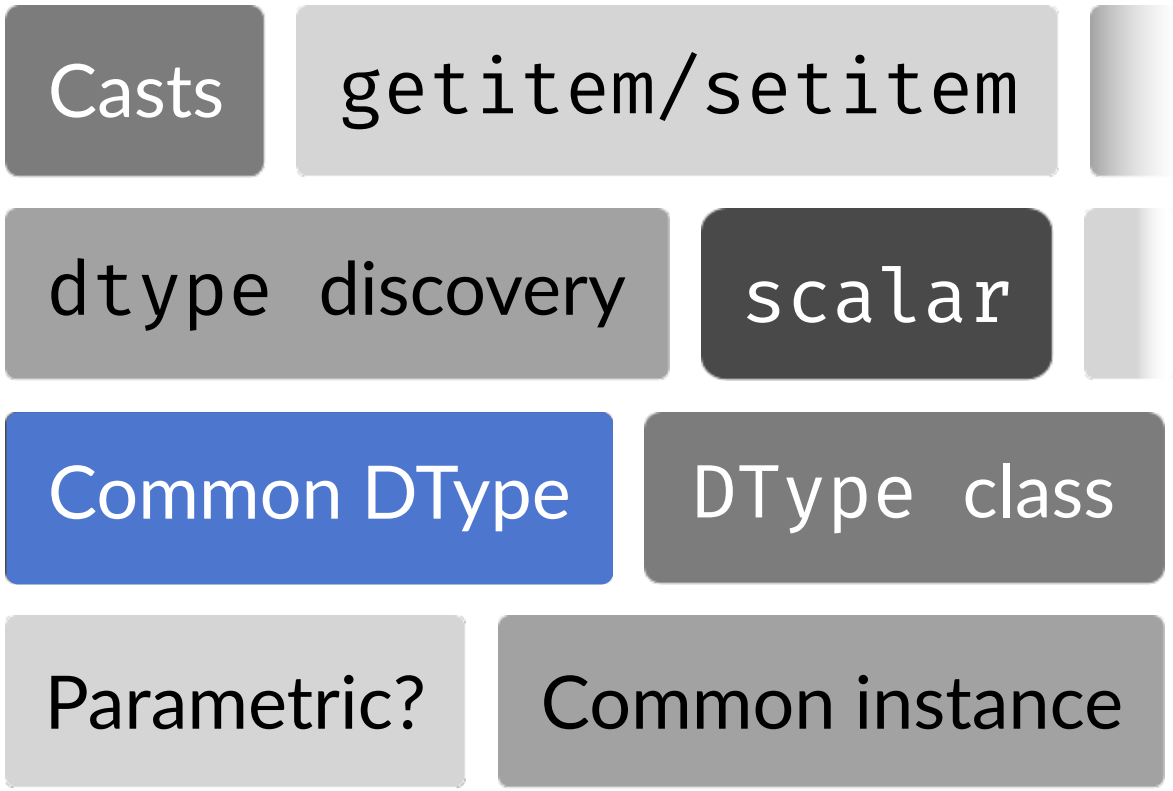


What makes the UnitDType?

Common `DType`:

- binary operator
- Promotion
- `concatenate()`

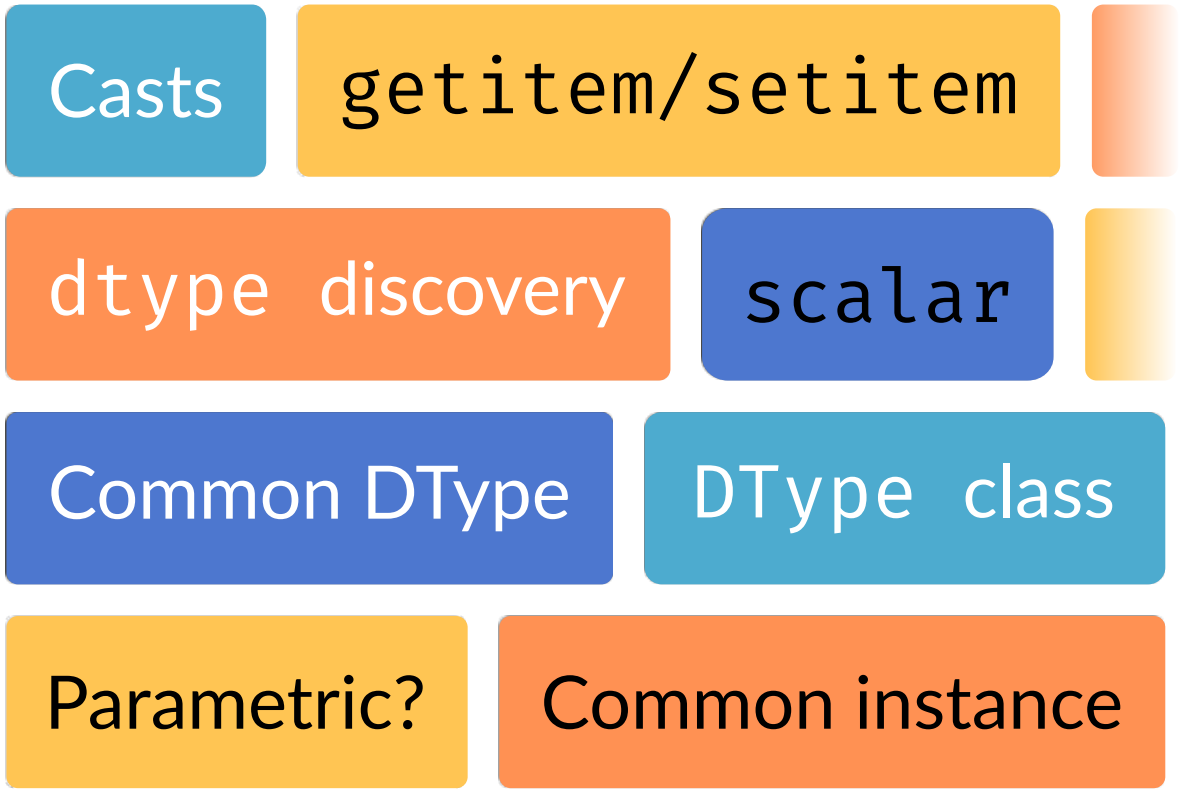
```
UnitDType & Float64
  → UnitDType
```



What makes the UnitDType?

Wrap it up:

- again: "`Spec`"
- extensible



Summary

- *First-class* user DTypes on the horizon
- Future extensibility easier
  - → e.g. new sorting methods
  - → string UFuncs
- experiment and help finalize API!

Acknowledgements:

Matti Picus, Marten van Kerkwijk, Ross Barnowsky, ...

This work was funded in part by the Gordon and Betty Moore Foundation through
Grant GBMF5447 and GBMF8599 and by the
Alfred P. Sloan Foundation through Grant G-2017-9960 to the University of California, Berkeley.

Summary

- *First-class* user DTypes on the horizon
- Future extensibility easier
  - → e.g. new sorting methods
  - → string UFuncs
- experiment and help finalize API!

```
numpy.org/neps  NEP 41-43
github.com/seberg/talk_unitdtype  (NumPy main)
github.com/seberg/unitdtype  (currently NumPy branch)
```

Thank You!