NumPy Advanced Cheat Sheet

1. Einstein Summation (‘einsum‘)

A concise and powerful way to express complex array operations like dot products, transpositions, and sums. The notation defines which axes are multiplied and which are summed over.

```
A = np.arange(6).reshape(2, 3)
B = np.arange(12).reshape(3, 4)

# Matrix multiplication (i=2, j=3, k=4)
# Sum over common dimension 'j'
# 'ij,jk->ik'
np.einsum('ij,jk', A, B)

# Get the main diagonal (trace)
C = np.arange(9).reshape(3, 3)
np.einsum('ii->i', C) # Returns: [0, 4, 8]

# Batch matrix multiplication
# b=batch, i=rows, j=common, k=cols
D = np.arange(24).reshape(2, 3, 4)
E = np.arange(32).reshape(2, 4, 2)
np.einsum('bij,bjk->bik', D, E) # (2,3,4) @ (2,4,2)
    -> (2,3,2)
```

2. Stride Tricks (‘as$_{s}$*trided*‘)

Create a view into an array with a specified shape and strides without copying data. **Warning:** Extremely powerful but dangerous. Incorrect strides can read garbage memory and crash your program.

Strides

A tuple of bytes to step in each dimension when traversing an array.

```
x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int32)
# itemsize is 4 bytes (for int32)
# To go to next row, jump 3*4=12 bytes
# To go to next col, jump 1*4=4 bytes
x.strides # Returns: (12, 4)
```

Example: Rolling Window

Create overlapping windows from a 1D array.

```
from numpy.lib.stride_tricks import as_strided

a = np.arange(10)
win_size = 4
itemsize = a.itemsize # Bytes per item

# Shape: (num_windows, window_size)
# Strides: (bytes_to_next_win, bytes_to_next_item)
rolling_view = as_strided(a,
    shape=(a.size - win_size + 1, win_size),
    strides=(itemsize, itemsize))
# [[0, 1, 2, 3],
#  [1, 2, 3, 4],
#  [2, 3, 4, 5], ... ]
```

3. Memory-Mapped Arrays (‘memmap‘)

Create arrays that reside on disk instead of in RAM. This allows you to work with datasets larger than your available memory. Changes are saved to the file.

```
# Create a new memory-mapped array on disk
fp = np.memmap('my_mmap.dat', dtype='float32',
               mode='w+', shape=(1000, 1000))

# The array is not in memory, but we can access it
fp[0, :] = 1.0 # Write a row
fp.flush() # Ensure changes are written to disk

# Load an existing memmap
fp_loaded = np.memmap('my_mmap.dat', dtype='float32'
    ,
                      mode='r', shape=(1000, 1000))
# fp_loaded[0, 0] is 1.0
```

4. Modern Random Generation (‘Generator‘)

The modern, recommended API for random numbers. It offers better statistical properties and is easier to use for reproducible science.

```
from numpy.random import default_rng

# Create a generator with a specific seed
# for reproducibility
rng = default_rng(seed=42)

# Generate 5 random integers from [0, 10)
rng.integers(low=0, high=10, size=5)

# Generate a 2x3 array from the standard
# normal distribution
rng.normal(size=(2, 3))

# Shuffle an array in-place
arr = np.arange(10)
rng.shuffle(arr)
```

5. Masked Arrays (‘numpy.ma‘)

Arrays that can have missing or invalid entries. Operations on these arrays automatically ignore the masked values.

```
import numpy.ma as ma

# Create an array with a missing value (nan)
x = np.array([1, 2, np.nan, 4])

# Create a masked array where nan is masked
mx = ma.masked_invalid(x)
# mx is: masked_array(data=[1., 2., --, 4.],
#                     mask=[False, False,  True,
    False]...)

# Operations ignore masked values
mx.mean() # Returns: 2.333 (sum of 1,2,4 / 3)

# Fill masked values with a constant
mx.filled(fill_value=0) # Returns: [1., 2., 0., 4.]
```

6. The ‘numpy.polynomial‘ package

A dedicated module for creating, manipulating, and fitting polynomials.

```
from numpy.polynomial import Polynomial as P

# Create a polynomial p(x) = 1 + 2x + 3x^2
# Coefficients are [c0, c1, c2, ...]
p = P([1, 2, 3])
```

NumPy Advanced Cheat Sheet

```
# Evaluate the polynomial at x=2
p(2) # Returns: 17.0 (1 + 2*2 + 3*2^2)

# Find the roots of the polynomial
p.roots() # Returns complex roots

# Find the derivative
p_deriv = p.deriv() # Is 2 + 6x

# Fit a polynomial of degree 2 to data
x = np.linspace(0, 1, 5)
y = 1 + 2*x + 3*x**2 + np.random.rand(5) * 0.1
p_fit = P.fit(x, y, deg=2)
```

7. Performance with JIT Compilation

For functions with loops that NumPy cannot vectorize, use **Numba** to compile Python code to fast, native machine code.

```
from numba import jit

# A Python loop that is hard to vectorize
def custom_sum(arr):
    total = 0
    for i in range(arr.size):
        if arr[i] > 0.5:
            total += arr[i]**2
    return total

# Apply the JIT decorator
# nopython=True ensures it runs at C speed
@jit(nopython=True)
def custom_sum_fast(arr):
    # (Same code as above)
    total = 0
    for i in range(arr.size):
        if arr[i] > 0.5:
            total += arr[i]**2
    return total

# The first run is slow (compilation time)
# Subsequent runs are extremely fast
# large_arr = rng.random(10**6)
# custom_sum_fast(large_arr)
```

8. Interoperability Protocols

NumPy defines protocols that allow other libraries (like CuPy, PyTorch, TensorFlow) to use NumPy functions on their own array-like objects.

‘$_{a}rray_{i}nterface_{.}$

A dictionary describing the array's memory layout, allowing zero-copy data exchange with other C-level libraries.

```
a = np.ones((2, 3), dtype=np.float64)
a.__array_interface__
# Returns a dict:
# {'shape': (2, 3),
#  'typestr': '<f8',
#  'data': (memory_address, False),
#  'version': 3,
#  'strides': None}
```

‘$_{a}rray_{f}unction_{.}$

Allows other libraries to override NumPy functions. When you call `np.sum(cupy_array)`, it dispatches to CuPy's implementation. You don't need to write code for this; it's how the ecosystem works.