

10.2_Numpy

October 30, 2023

1 Introduction to Python for Open Source Geocomputation



- Instructor: Dr. Wei Kang

Content:

- Numpy
- A new data type: `numpy.array`
 - How to create an array
 - Array operations

2 What is Numpy?

- The fundamental package for scientific computing with Python
- Nearly every scientist working in Python draws on the power of NumPy.
- NumPy brings the **computational power** of languages like C and Fortran to Python, a language much easier to learn and use. With this power comes **simplicity: a solution in NumPy is often clear and elegant.**
- Essential in many different realms:
 - NumPy lies at the core of a rich ecosystem of **data science** libraries
 - NumPy forms the basis of powerful **machine learning** libraries like [scikit-learn](#), [SciPy](#), [TensorFlow](#), and [PyTorch](#)
 - NumPy is an essential component in the burgeoning Python **visualization landscape**, which includes Matplotlib, Seaborn, Plotly, Altair, Bokeh, Holoviz, Vispy, Napari, and PyVista, to name a few.

2.1 What makes Numpy so important?

arrays: A very powerful data type essential to numerical computing: * sequences of data all of the *same type* * behave a lot like lists, except for the constraint in the type of their elements. * There is a huge efficiency advantage when you know that **all elements of a sequence are of the same type**—so equivalent methods for arrays execute a lot **faster** than those for lists.

2.2 Numpy Array (or ndarray)

- homogeneous multidimensional array
 - a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers
 - * for the data types accepted in Numpy. Read the [docs: Data type objects](#).
 - dimensions are called *axes*
- An Example: points' coordinates
 - one single point: one-dimensional array: `np.array([1,2])`
 - two or more points: two-dimensional array:
 - * two points: `np.array([[1,2], [3,4]])`
 - * five points: `np.array([[1,2], [3,4], [5,6], [7,8], [9,10]])`

```
[1]: import numpy as np
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.
```

```
[2]: a1 = np.array([1,2])
a1
```

```
[2]: array([1, 2])
```

```
[3]: a2 = np.array([[1,2], [3,4], [5,6], [7,8], [9,10]])
a2
```

```
[3]: array([[ 1,  2],
           [ 3,  4],
           [ 5,  6],
           [ 7,  8],
           [ 9, 10]])
```

2.2.1 Motivation (1): What can a Numpy array used for?

- An array can contain:
 - values of an experiment/simulation at discrete time steps, e.g., income, air pollution, crime rate, animal/plant occurrence

- pixels of an image, grey-level or colour
- signal recorded by a measurement device, e.g. sound wave
- 3-D data measured at different X-Y-Z positions, e.g. MRI scan, digital elevation model

2.2.2 Motivation (2): Efficiency of Numpy array - an example

- Problem description: Write a python program that calculate the square of each number in a list, such that $x_i = i^2$, for $0 \leq i < n$.

Two data types: * Python built-in type: list * Numpy array

We use `%timeit` to calculate the time execution of a Python statement or expression.

```
[4]: L = list(range(1000)) #produce a list of integers from 0 to 999
```

```
[5]: L
```

```
[5]: [0,
      1,
      2,
      3,
      4,
      5,
      6,
      7,
      8,
      9,
      10,
      11,
      12,
      13,
      14,
      15,
      16,
      17,
      18,
      19,
      20,
      21,
      22,
      23,
      24,
      25,
      26,
      27,
      28,
      29,
      30,
      31,
```

32,
33,
34,
35,
36,
37,
38,
39,
40,
41,
42,
43,
44,
45,
46,
47,
48,
49,
50,
51,
52,
53,
54,
55,
56,
57,
58,
59,
60,
61,
62,
63,
64,
65,
66,
67,
68,
69,
70,
71,
72,
73,
74,
75,
76,
77,
78,

79,
80,
81,
82,
83,
84,
85,
86,
87,
88,
89,
90,
91,
92,
93,
94,
95,
96,
97,
98,
99,
100,
101,
102,
103,
104,
105,
106,
107,
108,
109,
110,
111,
112,
113,
114,
115,
116,
117,
118,
119,
120,
121,
122,
123,
124,
125,

126,
127,
128,
129,
130,
131,
132,
133,
134,
135,
136,
137,
138,
139,
140,
141,
142,
143,
144,
145,
146,
147,
148,
149,
150,
151,
152,
153,
154,
155,
156,
157,
158,
159,
160,
161,
162,
163,
164,
165,
166,
167,
168,
169,
170,
171,
172,

173,
174,
175,
176,
177,
178,
179,
180,
181,
182,
183,
184,
185,
186,
187,
188,
189,
190,
191,
192,
193,
194,
195,
196,
197,
198,
199,
200,
201,
202,
203,
204,
205,
206,
207,
208,
209,
210,
211,
212,
213,
214,
215,
216,
217,
218,
219,

220,
221,
222,
223,
224,
225,
226,
227,
228,
229,
230,
231,
232,
233,
234,
235,
236,
237,
238,
239,
240,
241,
242,
243,
244,
245,
246,
247,
248,
249,
250,
251,
252,
253,
254,
255,
256,
257,
258,
259,
260,
261,
262,
263,
264,
265,
266,

267,
268,
269,
270,
271,
272,
273,
274,
275,
276,
277,
278,
279,
280,
281,
282,
283,
284,
285,
286,
287,
288,
289,
290,
291,
292,
293,
294,
295,
296,
297,
298,
299,
300,
301,
302,
303,
304,
305,
306,
307,
308,
309,
310,
311,
312,
313,

314,
315,
316,
317,
318,
319,
320,
321,
322,
323,
324,
325,
326,
327,
328,
329,
330,
331,
332,
333,
334,
335,
336,
337,
338,
339,
340,
341,
342,
343,
344,
345,
346,
347,
348,
349,
350,
351,
352,
353,
354,
355,
356,
357,
358,
359,
360,

361,
362,
363,
364,
365,
366,
367,
368,
369,
370,
371,
372,
373,
374,
375,
376,
377,
378,
379,
380,
381,
382,
383,
384,
385,
386,
387,
388,
389,
390,
391,
392,
393,
394,
395,
396,
397,
398,
399,
400,
401,
402,
403,
404,
405,
406,
407,

408,
409,
410,
411,
412,
413,
414,
415,
416,
417,
418,
419,
420,
421,
422,
423,
424,
425,
426,
427,
428,
429,
430,
431,
432,
433,
434,
435,
436,
437,
438,
439,
440,
441,
442,
443,
444,
445,
446,
447,
448,
449,
450,
451,
452,
453,
454,

455,
456,
457,
458,
459,
460,
461,
462,
463,
464,
465,
466,
467,
468,
469,
470,
471,
472,
473,
474,
475,
476,
477,
478,
479,
480,
481,
482,
483,
484,
485,
486,
487,
488,
489,
490,
491,
492,
493,
494,
495,
496,
497,
498,
499,
500,
501,

502,
503,
504,
505,
506,
507,
508,
509,
510,
511,
512,
513,
514,
515,
516,
517,
518,
519,
520,
521,
522,
523,
524,
525,
526,
527,
528,
529,
530,
531,
532,
533,
534,
535,
536,
537,
538,
539,
540,
541,
542,
543,
544,
545,
546,
547,
548,

549,
550,
551,
552,
553,
554,
555,
556,
557,
558,
559,
560,
561,
562,
563,
564,
565,
566,
567,
568,
569,
570,
571,
572,
573,
574,
575,
576,
577,
578,
579,
580,
581,
582,
583,
584,
585,
586,
587,
588,
589,
590,
591,
592,
593,
594,
595,

596,
597,
598,
599,
600,
601,
602,
603,
604,
605,
606,
607,
608,
609,
610,
611,
612,
613,
614,
615,
616,
617,
618,
619,
620,
621,
622,
623,
624,
625,
626,
627,
628,
629,
630,
631,
632,
633,
634,
635,
636,
637,
638,
639,
640,
641,
642,

643,
644,
645,
646,
647,
648,
649,
650,
651,
652,
653,
654,
655,
656,
657,
658,
659,
660,
661,
662,
663,
664,
665,
666,
667,
668,
669,
670,
671,
672,
673,
674,
675,
676,
677,
678,
679,
680,
681,
682,
683,
684,
685,
686,
687,
688,
689,

690,
691,
692,
693,
694,
695,
696,
697,
698,
699,
700,
701,
702,
703,
704,
705,
706,
707,
708,
709,
710,
711,
712,
713,
714,
715,
716,
717,
718,
719,
720,
721,
722,
723,
724,
725,
726,
727,
728,
729,
730,
731,
732,
733,
734,
735,
736,

737,
738,
739,
740,
741,
742,
743,
744,
745,
746,
747,
748,
749,
750,
751,
752,
753,
754,
755,
756,
757,
758,
759,
760,
761,
762,
763,
764,
765,
766,
767,
768,
769,
770,
771,
772,
773,
774,
775,
776,
777,
778,
779,
780,
781,
782,
783,

784,
785,
786,
787,
788,
789,
790,
791,
792,
793,
794,
795,
796,
797,
798,
799,
800,
801,
802,
803,
804,
805,
806,
807,
808,
809,
810,
811,
812,
813,
814,
815,
816,
817,
818,
819,
820,
821,
822,
823,
824,
825,
826,
827,
828,
829,
830,

831,
832,
833,
834,
835,
836,
837,
838,
839,
840,
841,
842,
843,
844,
845,
846,
847,
848,
849,
850,
851,
852,
853,
854,
855,
856,
857,
858,
859,
860,
861,
862,
863,
864,
865,
866,
867,
868,
869,
870,
871,
872,
873,
874,
875,
876,
877,

878,
879,
880,
881,
882,
883,
884,
885,
886,
887,
888,
889,
890,
891,
892,
893,
894,
895,
896,
897,
898,
899,
900,
901,
902,
903,
904,
905,
906,
907,
908,
909,
910,
911,
912,
913,
914,
915,
916,
917,
918,
919,
920,
921,
922,
923,
924,

925,
926,
927,
928,
929,
930,
931,
932,
933,
934,
935,
936,
937,
938,
939,
940,
941,
942,
943,
944,
945,
946,
947,
948,
949,
950,
951,
952,
953,
954,
955,
956,
957,
958,
959,
960,
961,
962,
963,
964,
965,
966,
967,
968,
969,
970,
971,

```
972,  
973,  
974,  
975,  
976,  
977,  
978,  
979,  
980,  
981,  
982,  
983,  
984,  
985,  
986,  
987,  
988,  
989,  
990,  
991,  
992,  
993,  
994,  
995,  
996,  
997,  
998,  
999]
```

```
[6]: L
```

```
[6]: [0,  
1,  
2,  
3,  
4,  
5,  
6,  
7,  
8,  
9,  
10,  
11,  
12,  
13,  
14,  
15,
```


16,
17,
18,
19,
20,
21,
22,
23,
24,
25,
26,
27,
28,
29,
30,
31,
32,
33,
34,
35,
36,
37,
38,
39,
40,
41,
42,
43,
44,
45,
46,
47,
48,
49,
50,
51,
52,
53,
54,
55,
56,
57,
58,
59,
60,
61,
62,

63,
64,
65,
66,
67,
68,
69,
70,
71,
72,
73,
74,
75,
76,
77,
78,
79,
80,
81,
82,
83,
84,
85,
86,
87,
88,
89,
90,
91,
92,
93,
94,
95,
96,
97,
98,
99,
100,
101,
102,
103,
104,
105,
106,
107,
108,
109,

110,
111,
112,
113,
114,
115,
116,
117,
118,
119,
120,
121,
122,
123,
124,
125,
126,
127,
128,
129,
130,
131,
132,
133,
134,
135,
136,
137,
138,
139,
140,
141,
142,
143,
144,
145,
146,
147,
148,
149,
150,
151,
152,
153,
154,
155,
156,

157,
158,
159,
160,
161,
162,
163,
164,
165,
166,
167,
168,
169,
170,
171,
172,
173,
174,
175,
176,
177,
178,
179,
180,
181,
182,
183,
184,
185,
186,
187,
188,
189,
190,
191,
192,
193,
194,
195,
196,
197,
198,
199,
200,
201,
202,
203,

204,
205,
206,
207,
208,
209,
210,
211,
212,
213,
214,
215,
216,
217,
218,
219,
220,
221,
222,
223,
224,
225,
226,
227,
228,
229,
230,
231,
232,
233,
234,
235,
236,
237,
238,
239,
240,
241,
242,
243,
244,
245,
246,
247,
248,
249,
250,

251,
252,
253,
254,
255,
256,
257,
258,
259,
260,
261,
262,
263,
264,
265,
266,
267,
268,
269,
270,
271,
272,
273,
274,
275,
276,
277,
278,
279,
280,
281,
282,
283,
284,
285,
286,
287,
288,
289,
290,
291,
292,
293,
294,
295,
296,
297,

298,
299,
300,
301,
302,
303,
304,
305,
306,
307,
308,
309,
310,
311,
312,
313,
314,
315,
316,
317,
318,
319,
320,
321,
322,
323,
324,
325,
326,
327,
328,
329,
330,
331,
332,
333,
334,
335,
336,
337,
338,
339,
340,
341,
342,
343,
344,

345,
346,
347,
348,
349,
350,
351,
352,
353,
354,
355,
356,
357,
358,
359,
360,
361,
362,
363,
364,
365,
366,
367,
368,
369,
370,
371,
372,
373,
374,
375,
376,
377,
378,
379,
380,
381,
382,
383,
384,
385,
386,
387,
388,
389,
390,
391,

392,
393,
394,
395,
396,
397,
398,
399,
400,
401,
402,
403,
404,
405,
406,
407,
408,
409,
410,
411,
412,
413,
414,
415,
416,
417,
418,
419,
420,
421,
422,
423,
424,
425,
426,
427,
428,
429,
430,
431,
432,
433,
434,
435,
436,
437,
438,

439,
440,
441,
442,
443,
444,
445,
446,
447,
448,
449,
450,
451,
452,
453,
454,
455,
456,
457,
458,
459,
460,
461,
462,
463,
464,
465,
466,
467,
468,
469,
470,
471,
472,
473,
474,
475,
476,
477,
478,
479,
480,
481,
482,
483,
484,
485,

486,
487,
488,
489,
490,
491,
492,
493,
494,
495,
496,
497,
498,
499,
500,
501,
502,
503,
504,
505,
506,
507,
508,
509,
510,
511,
512,
513,
514,
515,
516,
517,
518,
519,
520,
521,
522,
523,
524,
525,
526,
527,
528,
529,
530,
531,
532,

533,
534,
535,
536,
537,
538,
539,
540,
541,
542,
543,
544,
545,
546,
547,
548,
549,
550,
551,
552,
553,
554,
555,
556,
557,
558,
559,
560,
561,
562,
563,
564,
565,
566,
567,
568,
569,
570,
571,
572,
573,
574,
575,
576,
577,
578,
579,

580,
581,
582,
583,
584,
585,
586,
587,
588,
589,
590,
591,
592,
593,
594,
595,
596,
597,
598,
599,
600,
601,
602,
603,
604,
605,
606,
607,
608,
609,
610,
611,
612,
613,
614,
615,
616,
617,
618,
619,
620,
621,
622,
623,
624,
625,
626,

627,
628,
629,
630,
631,
632,
633,
634,
635,
636,
637,
638,
639,
640,
641,
642,
643,
644,
645,
646,
647,
648,
649,
650,
651,
652,
653,
654,
655,
656,
657,
658,
659,
660,
661,
662,
663,
664,
665,
666,
667,
668,
669,
670,
671,
672,
673,

674,
675,
676,
677,
678,
679,
680,
681,
682,
683,
684,
685,
686,
687,
688,
689,
690,
691,
692,
693,
694,
695,
696,
697,
698,
699,
700,
701,
702,
703,
704,
705,
706,
707,
708,
709,
710,
711,
712,
713,
714,
715,
716,
717,
718,
719,
720,

721,
722,
723,
724,
725,
726,
727,
728,
729,
730,
731,
732,
733,
734,
735,
736,
737,
738,
739,
740,
741,
742,
743,
744,
745,
746,
747,
748,
749,
750,
751,
752,
753,
754,
755,
756,
757,
758,
759,
760,
761,
762,
763,
764,
765,
766,
767,

768,
769,
770,
771,
772,
773,
774,
775,
776,
777,
778,
779,
780,
781,
782,
783,
784,
785,
786,
787,
788,
789,
790,
791,
792,
793,
794,
795,
796,
797,
798,
799,
800,
801,
802,
803,
804,
805,
806,
807,
808,
809,
810,
811,
812,
813,
814,

815,
816,
817,
818,
819,
820,
821,
822,
823,
824,
825,
826,
827,
828,
829,
830,
831,
832,
833,
834,
835,
836,
837,
838,
839,
840,
841,
842,
843,
844,
845,
846,
847,
848,
849,
850,
851,
852,
853,
854,
855,
856,
857,
858,
859,
860,
861,

862,
863,
864,
865,
866,
867,
868,
869,
870,
871,
872,
873,
874,
875,
876,
877,
878,
879,
880,
881,
882,
883,
884,
885,
886,
887,
888,
889,
890,
891,
892,
893,
894,
895,
896,
897,
898,
899,
900,
901,
902,
903,
904,
905,
906,
907,
908,

909,
910,
911,
912,
913,
914,
915,
916,
917,
918,
919,
920,
921,
922,
923,
924,
925,
926,
927,
928,
929,
930,
931,
932,
933,
934,
935,
936,
937,
938,
939,
940,
941,
942,
943,
944,
945,
946,
947,
948,
949,
950,
951,
952,
953,
954,
955,

956,
957,
958,
959,
960,
961,
962,
963,
964,
965,
966,
967,
968,
969,
970,
971,
972,
973,
974,
975,
976,
977,
978,
979,
980,
981,
982,
983,
984,
985,
986,
987,
988,
989,
990,
991,
992,
993,
994,
995,
996,
997,
998,
999]

```
[7]: L = list(range(1000))  
     for i in range(len(L)):
```

```
L[i] = L[i] **2
```

list comprehension

```
[8]: %timeit -n 1000 [i**2 for i in L]
```

184 μ s \pm 3.53 μ s per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)

```
[9]: import numpy as np
a = np.arange(1000) #produce an array of integers from 0 to 999
# a
```

```
[10]: %timeit -n 1000 a**2
```

1.11 μ s \pm 280 ns per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)

3 Importing Numpy

import numpy as np

```
[11]: import numpy as np
```

```
[12]: dir(np) #function dir gives you the package's attributes and functions.
```

```
[12]: ['ALLOW_THREADS',
      'AxisError',
      'BUFSIZE',
      'CLIP',
      'ComplexWarning',
      'DataSource',
      'ERR_CALL',
      'ERR_DEFAULT',
      'ERR_IGNORE',
      'ERR_LOG',
      'ERR_PRINT',
      'ERR_RAISE',
      'ERR_WARN',
      'FLOATING_POINT_SUPPORT',
      'FPE_DIVIDEBYZERO',
      'FPE_INVALID',
      'FPE_OVERFLOW',
      'FPE_UNDERFLOW',
      'False_',
      'Inf',
      'Infinity',
      'MAXDIMS',
      'MAY_SHARE_BOUNDS',
      'MAY_SHARE_EXACT',
```

```

'ModuleDeprecationWarning',
'NaN',
'NINF',
'NZERO',
'NaN',
'PINF',
'PZERO',
'RAISE',
'RankWarning',
'SHIFT_DIVIDEBYZERO',
'SHIFT_INVALID',
'SHIFT_OVERFLOW',
'SHIFT_UNDERFLOW',
'ScalarType',
'Tester',
'TooHardError',
'True_',
'UFUNC_BUFSIZE_DEFAULT',
'UFUNC_PYVALS_NAME',
'VisibleDeprecationWarning',
'WRAP',
'_CopyMode',
'_NoValue',
'_UFUNC_API',
'__NUMPY_SETUP__',
'__all__',
'__builtins__',
'__cached__',
'__config__',
'__deprecated_attrs__',
'__dir__',
'__doc__',
'__expired_functions__',
'__file__',
'__former_attrs__',
'__future_scalars__',
'__getattr__',
'__git_version__',
'__loader__',
'__mkl_version__',
'__name__',
'__package__',
'__path__',
'__spec__',
'__version__',
'_add_newdoc_ufunc',
'_builtins',

```

'_distributor_init',
'_financial_names',
'_get_promotion_state',
'_globals',
'_int_extended_msg',
'_mat',
'_no_nep50_warning',
'_pyinstaller_hooks_dir',
'_pytesttester',
'_set_promotion_state',
'_specific_msg',
'_version',
'abs',
'absolute',
'add',
'add_docstring',
'add_newdoc',
'add_newdoc_ufunc',
'all',
'allclose',
'alltrue',
'amax',
'amin',
'angle',
'any',
'append',
'apply_along_axis',
'apply_over_axes',
'arange',
'arccos',
'arccosh',
'arcsin',
'arcsinh',
'arctan',
'arctan2',
'arctanh',
'argmax',
'argmin',
'argpartition',
'argsort',
'argwhere',
'around',
'array',
'array2string',
'array_equal',
'array_equiv',
'array_repr',

'array_split',
'array_str',
'asanyarray',
'asarray',
'asarray_chkfinite',
'ascontiguousarray',
'asfarray',
'asfortranarray',
'asmatrix',
'atleast_1d',
'atleast_2d',
'atleast_3d',
'average',
'bartlett',
'base_repr',
'binary_repr',
'bincount',
'bitwise_and',
'bitwise_not',
'bitwise_or',
'bitwise_xor',
'blackman',
'block',
'bmat',
'bool_',
'broadcast',
'broadcast_arrays',
'broadcast_shapes',
'broadcast_to',
'busday_count',
'busday_offset',
'busdaycalendar',
'byte',
'byte_bounds',
'bytes_',
'c_',
'can_cast',
'cast',
'cbrt',
'cdouble',
'ceil',
'cfloat',
'char',
'character',
'chararray',
'choose',
'clip',

'clongdouble',
'clongfloat',
'column_stack',
'common_type',
'compare_chararrays',
'compat',
'complex128',
'complex256',
'complex64',
'complex_',
'complexfloating',
'compress',
'concatenate',
'conj',
'conjugate',
'convolve',
'copy',
'copysign',
'copyto',
'corrcoef',
'correlate',
'cos',
'cosh',
'count_nonzero',
'cov',
'cross',
'csingle',
'ctypeslib',
'cumprod',
'cumproduct',
'cumsum',
'datetime64',
'datetime_as_string',
'datetime_data',
'deg2rad',
'degrees',
'delete',
'deprecate',
'deprecate_with_doc',
'diag',
'diag_indices',
'diag_indices_from',
'diagflat',
'diagonal',
'diff',
'digitize',
'disp',

'divide',
'divmod',
'dot',
'double',
'dsplit',
'dstack',
'dtype',
'e',
'ediff1d',
'einsum',
'einsum_path',
'emath',
'empty',
'empty_like',
'equal',
'error_message',
'errstate',
'euler_gamma',
'exp',
'exp2',
'expand_dims',
'expm1',
'extract',
'eye',
'fabs',
'fastCopyAndTranspose',
'fft',
'fill_diagonal',
'find_common_type',
'finfo',
'fix',
'flatiter',
'flatnonzero',
'flexible',
'flip',
'fliplr',
'flipud',
'float128',
'float16',
'float32',
'float64',
'float_',
'float_power',
'floating',
'floor',
'floor_divide',
'fmax',

'fmin',
'fmod',
'format_float_positional',
'format_float_scientific',
'format_parser',
'frexp',
'from_dlpack',
'frombuffer',
'fromfile',
'fromfunction',
'fromiter',
'frompyfunc',
'fromregex',
'fromstring',
'full',
'full_like',
'gcd',
'generic',
'genfromtxt',
'geomspace',
'get_array_wrap',
'get_include',
'get_printoptions',
'getbufsize',
'geterr',
'geterrcall',
'geterrobj',
'gradient',
'greater',
'greater_equal',
'half',
'hamming',
'hanning',
'heaviside',
'histogram',
'histogram2d',
'histogram_bin_edges',
'histogramdd',
'hsplit',
'hstack',
'hypot',
'i0',
'identity',
'iinfo',
'imag',
'in1d',
'index_exp',

'indices',
'inexact',
'inf',
'info',
'infty',
'inner',
'insert',
'int16',
'int32',
'int64',
'int8',
'int_',
'intc',
'integer',
'interp',
'intersect1d',
'intp',
'invert',
'is_busday',
'isclose',
'iscomplex',
'iscomplexobj',
'isfinite',
'isfortran',
'isin',
'isinf',
'isnan',
'isnat',
'isneginf',
'isposinf',
'isreal',
'isrealobj',
'isscalar',
'issctype',
'issubclass_',
'issubdtype',
'issubdtype',
'iterable',
'ix_',
'kaiser',
'kron',
'lcm',
'ldexp',
'left_shift',
'less',
'less_equal',
'lexsort',

'lib',
'linalg',
'linspace',
'little_endian',
'load',
'loadtxt',
'log',
'log10',
'log1p',
'log2',
'logaddexp',
'logaddexp2',
'logical_and',
'logical_not',
'logical_or',
'logical_xor',
'logspace',
'longcomplex',
'longdouble',
'longfloat',
'longlong',
'lookfor',
'ma',
'mask_indices',
'mat',
'math',
'matmul',
'matrix',
'max',
'maximum',
'maximum_sctype',
'may_share_memory',
'mean',
'median',
'memmap',
'meshgrid',
'mgrid',
'min',
'min_scalar_type',
'minimum',
'mintypecode',
'mkl',
'mod',
'modf',
'moveaxis',
'msort',
'multiply',

'nan',
'nan_to_num',
'nanargmax',
'nanargmin',
'nancumprod',
'nancumsum',
'nanmax',
'nanmean',
'nanmedian',
'nanmin',
'nanpercentile',
'nanprod',
'nanquantile',
'nanstd',
'nansum',
'nanvar',
'nbytes',
'ndarray',
'ndenumerate',
'ndim',
'ndindex',
'nditer',
'negative',
'nested_iters',
'newaxis',
'nextafter',
'nonzero',
'not_equal',
'numarray',
'number',
'obj2sctype',
'object_',
'ogrid',
'oldnumeric',
'ones',
'ones_like',
'outer',
'packbits',
'pad',
'partition',
'percentile',
'pi',
'piecewise',
'place',
'poly',
'poly1d',
'polyadd',

'polyder',
'polydiv',
'polyfit',
'polyint',
'polymul',
'polynomial',
'polysub',
'polyval',
'positive',
'power',
'printoptions',
'prod',
'product',
'promote_types',
'ptp',
'put',
'put_along_axis',
'putmask',
'quantile',
'r_',
'rad2deg',
'radians',
'random',
'ravel',
'ravel_multi_index',
'real',
'real_if_close',
'rec',
'recarray',
'recfromcsv',
'recfromtxt',
'reciprocal',
'record',
'remainder',
'repeat',
'require',
'reshape',
'resize',
'result_type',
'right_shift',
'rint',
'roll',
'rollaxis',
'roots',
'rot90',
'round',
'round_',

'row_stack',
's_',
'safe_eval',
'save',
'savetxt',
'savez',
'savez_compressed',
'sctype2char',
'sctypeDict',
'sctypes',
'searchsorted',
'select',
'set_numeric_ops',
'set_printoptions',
'set_string_function',
'setbufsize',
'setdiff1d',
'seterr',
'seterrcall',
'seterrobj',
'setxor1d',
'shape',
'shares_memory',
'short',
'show_config',
'show_runtime',
'sign',
'signbit',
'signedinteger',
'sin',
'sinc',
'single',
'singlecomplex',
'sinh',
'size',
'sometrue',
'sort',
'sort_complex',
'source',
'spacing',
'split',
'sqrt',
'square',
'squeeze',
'stack',
'std',
'str_',

'string_',
'subtract',
'sum',
'swapaxes',
'take',
'take_along_axis',
'tan',
'tanh',
'tensordot',
'test',
'testing',
'tile',
'timedelta64',
'trace',
'tracemalloc_domain',
'transpose',
'trapz',
'tri',
'tril',
'tril_indices',
'tril_indices_from',
'trim_zeros',
'triu',
'triu_indices',
'triu_indices_from',
'true_divide',
'trunc',
'typecodes',
'typename',
'ubyte',
'ufunc',
'uint',
'uint16',
'uint32',
'uint64',
'uint8',
'uintc',
'uintp',
'ulonglong',
'unicode_',
'union1d',
'unique',
'unpackbits',
'unravel_index',
'unsignedinteger',
'unwrap',
'use_hugepage',

```
'ushort',
'vander',
'var',
'vdot',
'vectorize',
'version',
'void',
'vsplit',
'vstack',
'w',
'where',
'who',
'zeros',
'zeros_like']
```

3.0.1 Creating a Numpy Array

- create an array from a regular Python list or tuple using the `array` function.

`np.array(list/tuple)`

- functions from Numpy to create special arrays
 - `np.arange()`: create evenly spaced values within a given interval.
 - `np.linspace(start, stop, num=50)`: create evenly spaced numbers over a specified interval.
 - `np.ones(shape)`: create new array of given shape and type, filled with ones.
 - `np.zeros(shape)`: create a new array of given shape and type, filled with zeros.
 - `np.eye(N)`: create a 2-D array with ones on the diagonal and zeros elsewhere.

```
[13]: a1 = np.array([1,2])
a1
```

```
[13]: array([1, 2])
```

```
[14]: type(a1)
```

```
[14]: numpy.ndarray
```

```
[15]: a1.size
```

```
[15]: 2
```

`array.size` gives the number of items in the array.

```
[16]: len(a1)
```

```
[16]: 2
```

`len(array)` gives the same result to `array.size`

```
[17]: a1.ndim
```

```
[17]: 1
```

`array.ndim` gives the number of axes (dimensions) of the array.

```
[18]: a1.shape
```

```
[18]: (2,)
```

```
[19]: a1
```

```
[19]: array([1, 2])
```

`array.shape` gives the dimensions of the array. This is a tuple of integers indicating the **size** of the array in each dimension. For a matrix with *n* rows and *m* columns, shape will be (*n*,*m*). The length of the shape tuple is therefore the number of axes, `ndim`.

```
[20]: a1.dtype
```

```
[20]: dtype('int64')
```

`array.dtype` returns an object describing the type of the elements in the array

```
[21]: a_str = np.array([1.0,2,"1"])
a_str
```

```
[21]: array(['1.0', '2', '1'], dtype='<U32')
```

```
[22]: a_str.dtype #32-character string
```

```
[22]: dtype('<U32')
```

```
[23]: a2 = np.array([[1,2], [3,4]])
a2
```

```
[23]: array([[1, 2],
          [3, 4]])
```

```
[24]: a2.ndim
```

```
[24]: 2
```

```
[25]: a2.size
```

```
[25]: 4
```

```
[26]: len(a2)
```

```
[26]: 2
```

`len(array)` gives the number of rows or the size of the first dimension when encountering a 2-dimensional array

```
[27]: a2.shape
```

```
[27]: (2, 2)
```

```
[28]: a2.dtype
```

```
[28]: dtype('int64')
```

```
[29]: a3 = np.array([[1,2], [3,4],[5,6], [7,8], [9,10]])  
a3
```

```
[29]: array([[ 1,  2],  
           [ 3,  4],  
           [ 5,  6],  
           [ 7,  8],  
           [ 9, 10]])
```

```
[30]: a3.ndim
```

```
[30]: 2
```

```
[31]: len(a3)
```

```
[31]: 5
```

```
[32]: a3.size
```

```
[32]: 10
```

```
[33]: a3
```

```
[33]: array([[ 1,  2],  
           [ 3,  4],  
           [ 5,  6],  
           [ 7,  8],  
           [ 9, 10]])
```

```
[34]: a3.shape
```

```
[34]: (5, 2)
```

```
[35]: a3.dtype
```

```
[35]: dtype('int64')
```

We can create a 3-dimensional array

```
[36]: import numpy as np
```

```
[37]: c = np.array([[1,1], [2,2]],  
                  [[3,23], [4,5]],  
                  [[5,3], [9,10]])  
c
```

```
[37]: array([[[ 1,  1],  
             [ 2,  2]],  
           [[ 3, 23],  
            [ 4,  5]],  
           [[ 5,  3],  
            [ 9, 10]]])
```

```
[38]: c.ndim
```

```
[38]: 3
```

```
[39]: c.shape
```

```
[39]: (3, 2, 2)
```

```
[40]: c.size
```

```
[40]: 12
```

```
[41]: len(c)
```

```
[41]: 3
```

```
[42]: a = np.array(1, 2, 3, 4)
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[42], line 1  
----> 1 a = np.array(1, 2, 3, 4)  
  
TypeError: array() takes from 1 to 2 positional arguments but 4 were given
```

The input needs to be an ordered sequence data type: list or tuples

```
[43]: a = np.array([1, 2, 3, 4])  
a
```

```
[43]: array([1, 2, 3, 4])
```

```
[44]: a = np.array((1, 2, 3, 4))
a
```

```
[44]: array([1, 2, 3, 4])
```

```
[45]: a = np.array((1, 2, 3, 4), (1, 2, 3, 4))
a
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[45], line 1
----> 1 a = np.array((1, 2, 3, 4), (1, 2, 3, 4))
      2 a

TypeError: Tuple must have size 2, but has size 4
```

```
[46]: a = np.array(((1, 2, 3, 4), (1, 2, 3, 4)))
a
```

```
[46]: array([[1, 2, 3, 4],
          [1, 2, 3, 4]])
```

```
[47]: a = np.array([[1, 2, 3, 4], [1, 2, 3, 4]])
a
```

```
[47]: array([[1, 2, 3, 4],
          [1, 2, 3, 4]])
```

3.1 Numpy functions to generate special arrays

3.1.1 numpy.arange()

`numpy.arange()` gives an array of evenly spaced values in a defined interval. Similar to `range()`

Syntax:

`numpy.arange(start, stop, step)`

where `start` by default is zero, `stop` is not inclusive, and the default for `step` is one.

```
[48]: list(range(3))
```

```
[48]: [0, 1, 2]
```

```
[49]: np.arange(3) # 0 .. n-1 (!)
```

```
[49]: array([0, 1, 2])
```

```
[50]: np.arange(2, 6)
```

```
[50]: array([2, 3, 4, 5])
```

```
[51]: np.arange(2, 6, 2) # start, end (exclusive), step
```

```
[51]: array([2, 4])
```

```
[52]: np.arange(2, 6, 0.5) # start, end (exclusive), step
```

```
[52]: array([2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5])
```

3.1.2 numpy.linspace()

`numpy.linspace()` is similar to `numpy.arange()`, but uses number of samples instead of a step size. It returns an array with evenly spaced numbers over the specified interval.

Syntax:

```
numpy.linspace(start, stop, num)
```

`stop` is included by default (it can be removed, read the docs), and `num` by default is 50.

```
[53]: np.linspace(2.0, 3.0)
```

```
[53]: array([2.          , 2.02040816, 2.04081633, 2.06122449, 2.08163265,
          2.10204082, 2.12244898, 2.14285714, 2.16326531, 2.18367347,
          2.20408163, 2.2244898 , 2.24489796, 2.26530612, 2.28571429,
          2.30612245, 2.32653061, 2.34693878, 2.36734694, 2.3877551 ,
          2.40816327, 2.42857143, 2.44897959, 2.46938776, 2.48979592,
          2.51020408, 2.53061224, 2.55102041, 2.57142857, 2.59183673,
          2.6122449 , 2.63265306, 2.65306122, 2.67346939, 2.69387755,
          2.71428571, 2.73469388, 2.75510204, 2.7755102 , 2.79591837,
          2.81632653, 2.83673469, 2.85714286, 2.87755102, 2.89795918,
          2.91836735, 2.93877551, 2.95918367, 2.97959184, 3.          ])
```

```
[54]: len(np.linspace(2.0, 3.0))
```

```
[54]: 50
```

```
[55]: np.linspace(0, 1, 6) # start, end, num of points
```

```
[55]: array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])
```

```
[56]: np.linspace(-1, 1, 9)
```

```
[56]: array([-1. , -0.75, -0.5 , -0.25,  0. ,  0.25,  0.5 ,  0.75,  1. ])
```

We can also create special arrays using Numpy functions

```
[57]: a = np.ones(3) # creating a 1-D array full of 1s
```

```
[58]: a
```



```
[58]: array([1., 1., 1.])
```

```
[59]: a = np.ones((3, 2)) # (3,2) is the shape of the array we want to create, which  
      ↪ needs to be a tuple
```

```
[60]: a
```

```
[60]: array([[1., 1.],  
           [1., 1.],  
           [1., 1.]])
```

```
[61]: b = np.zeros(10) # creating a 1-D array full of 0s
```

```
[62]: b
```

```
[62]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
[63]: b = np.zeros((3, 3))
```

```
[64]: b
```

```
[64]: array([[0., 0., 0.],  
           [0., 0., 0.],  
           [0., 0., 0.]])
```

```
[65]: np.eye(5)
```

```
[65]: array([[1., 0., 0., 0., 0.],  
           [0., 1., 0., 0., 0.],  
           [0., 0., 1., 0., 0.],  
           [0., 0., 0., 1., 0.],  
           [0., 0., 0., 0., 1.]])
```

```
[66]: np.eye(2)
```

```
[66]: array([[1., 0.],  
           [0., 1.]])
```

```
[67]: np.empty((2, 3))
```

```
[67]: array([[1., 1., 1.],  
           [1., 1., 1.]])
```

3.2 Arithmetic operations on arrays

- Arithmetic operators on arrays apply elementwise
- Different from the application of Arithmetic operators to lists

```
[68]: a = np.array([20, 30, 40, 50])  
      b = np.array([1,2,3,4])
```

```
[69]: c = a + b  
      c
```

```
[69]: array([21, 32, 43, 54])
```

```
[70]: d = np.array([1,2,3])
```

```
[71]: a + d
```

```
-----  
ValueError                                Traceback (most recent call last)  
Cell In[71], line 1  
----> 1 a + d  
  
ValueError: operands could not be broadcast together with shapes (4,) (3,)
```

```
[72]: list_a = list(a)  
      list_b = list(b)
```

```
[73]: type(list_a)
```

```
[73]: list
```

```
[74]: list_a
```

```
[74]: [20, 30, 40, 50]
```

```
[75]: list_b
```

```
[75]: [1, 2, 3, 4]
```

```
[76]: list_a + list_b
```

```
[76]: [20, 30, 40, 50, 1, 2, 3, 4]
```

```
[77]: a
```

```
[77]: array([20, 30, 40, 50])
```

```
[78]: b
```

```
[78]: array([1, 2, 3, 4])
```

```
[79]: d = a - b
      d
```

```
[79]: array([19, 28, 37, 46])
```

```
[80]: list_d = list_a - list_b
      list_d
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[80], line 1
----> 1 list_d = list_a - list_b
      2 list_d

TypeError: unsupported operand type(s) for -: 'list' and 'list'
```

```
[81]: a
```

```
[81]: array([20, 30, 40, 50])
```

```
[82]: a ** 2
```

```
[82]: array([ 400,  900, 1600, 2500])
```

```
[83]: list_a ** 2
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[83], line 1
----> 1 list_a ** 2

TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

```
[84]: a
```

```
[84]: array([20, 30, 40, 50])
```

```
[85]: a * 2
```

```
[85]: array([ 40,  60,  80, 100])
```

```
[86]: list_a
```

```
[86]: [20, 30, 40, 50]
```

```
[87]: list_a * 2
```

```
[87]: [20, 30, 40, 50, 20, 30, 40, 50]
```

```
[88]: a
```

```
[88]: array([20, 30, 40, 50])
```

```
[89]: b
```

```
[89]: array([1, 2, 3, 4])
```

```
[90]: a < b
```

```
[90]: array([False, False, False, False])
```

```
[91]: list_a
```

```
[91]: [20, 30, 40, 50]
```

```
[92]: list_b
```

```
[92]: [1, 2, 3, 4]
```

```
[93]: list_a < list_b
```

```
[93]: False
```

```
[94]: a
```

```
[94]: array([20, 30, 40, 50])
```

```
[95]: b
```

```
[95]: array([1, 2, 3, 4])
```

```
[96]: a/b
```

```
[96]: array([20.         , 15.         , 13.33333333, 12.5         ])
```

```
[97]: list_a/list_b
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[97], line 1  
----> 1 list_a/list_b  
  
TypeError: unsupported operand type(s) for /: 'list' and 'list'
```

```
[98]: a.shape
```

```
[98]: (4,)
```

```
[99]: a
```

```
[99]: array([20, 30, 40, 50])
```

```
[100]: a + 1
```

```
[100]: array([21, 31, 41, 51])
```

Broadcasting with scalar numerical data type

```
[101]: list_a + 1
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[101], line 1  
----> 1 list_a + 1  
  
TypeError: can only concatenate list (not "int") to list
```

```
[102]: a
```

```
[102]: array([20, 30, 40, 50])
```

```
[103]: a < 30
```

```
[103]: array([ True, False, False, False])
```

```
[104]: list_a < 30
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[104], line 1  
----> 1 list_a < 30  
  
TypeError: '<' not supported between instances of 'list' and 'int'
```

```
[105]: c = np.array([10,15,20])
```

```
[106]: a
```

```
[106]: array([20, 30, 40, 50])
```

```
[107]: a + c
```

```

-----
ValueError                                Traceback (most recent call last)
Cell In[107], line 1
----> 1 a + c

ValueError: operands could not be broadcast together with shapes (4,) (3,)

```

Shape mismatches

Arithmetic operation on 2-D arrays

```
[108]: X = np.array([[1, 2], [3, 4]])
print(X)
```

```
[[1 2]
 [3 4]]
```

```
[109]: Y = np.array([[1, -1], [0, 1]])
print(Y)
```

```
[[ 1 -1]
 [ 0  1]]
```

```
[110]: X + Y
```

```
[110]: array([[2, 1],
             [3, 5]])
```

```
[111]: X * Y
```

```
[111]: array([[ 1, -2],
             [ 0,  4]])
```

The multiplication using the '*' operator is element-wise.

What if we want to do matrix multiplication? Using the '@' operator:

"Dot Product"

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 \\ \end{bmatrix}$$

```
[112]: X * Y
```

```
[112]: array([[ 1, -2],  
            [ 0,  4]])
```

```
[113]: X @ Y
```

```
[113]: array([[1, 1],  
            [3, 1]])
```

Or equivalently we can use `np.dot()`:

```
[114]: np.dot(X, Y)
```

```
[114]: array([[1, 1],  
            [3, 1]])
```

```
[115]: Z = np.arange(12)  
Z
```

```
[115]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
[116]: Z.reshape(3,4) #reshape() change the shape of an array
```

```
[116]: array([[ 0,  1,  2,  3],  
            [ 4,  5,  6,  7],  
            [ 8,  9, 10, 11]])
```

```
[117]: Z.reshape((3,4)) #reshape() change the shape of an array
```

```
[117]: array([[ 0,  1,  2,  3],  
            [ 4,  5,  6,  7],  
            [ 8,  9, 10, 11]])
```

```
[118]: Z = Z.reshape(3,4)  
Z
```

```
[118]: array([[ 0,  1,  2,  3],  
            [ 4,  5,  6,  7],  
            [ 8,  9, 10, 11]])
```

```
[119]: Z.sum()
```

```
[119]: 66
```

```
[120]: Z.max()
```

```
[120]: 11
```

```
[121]: Z.min()
```

```
[121]: 0
```

```
[122]: Z
```

```
[122]: array([[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11]])
```

```
[123]: Z.sum(axis=0) # sum of each column
```

```
[123]: array([12, 15, 18, 21])
```

```
[124]: Z.sum(axis=1) # sum of each row
```

```
[124]: array([ 6, 22, 38])
```

```
[125]: Z.mean(axis=1) # average of each row
```

```
[125]: array([1.5, 5.5, 9.5])
```

```
[126]: Z.mean(axis=0) # average of each column
```

```
[126]: array([4., 5., 6., 7.])
```

3.3 Indexing, Slicing and Iterating

- 1-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.
- Multidimensional arrays have one index per axis. These indices are given in a tuple separated by commas

```
[127]: a = np.arange(10)**3
a
```

```
[127]: array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
```

```
[128]: a[0]
```

```
[128]: 0
```

```
[129]: a[3]
```

```
[129]: 27
```

```
[130]: a[2:5]
```

```
[130]: array([ 8, 27, 64])
```



```
[131]: for i in a:  
        print(i)
```

```
0  
1  
8  
27  
64  
125  
216  
343  
512  
729
```

```
[132]: b = np.arange(12).reshape(3,4)  
b
```

```
[132]: array([[ 0,  1,  2,  3],  
            [ 4,  5,  6,  7],  
            [ 8,  9, 10, 11]])
```

```
[133]: list_b = []  
for i in b:  
    list_b.append(list(i))  
list_b
```

```
[133]: [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

```
[134]: list_b[0][0]
```

```
[134]: 0
```

```
[135]: b
```

```
[135]: array([[ 0,  1,  2,  3],  
            [ 4,  5,  6,  7],  
            [ 8,  9, 10, 11]])
```

```
[136]: b[0,0]
```

```
[136]: 0
```

```
[137]: b
```

```
[137]: array([[ 0,  1,  2,  3],  
            [ 4,  5,  6,  7],  
            [ 8,  9, 10, 11]])
```

```
[138]: b[2,3]
```

```
[138]: 11
```

```
[139]: b[:2,0]
```

```
[139]: array([0, 4])
```

```
[140]: b[:,0]
```

```
[140]: array([0, 4, 8])
```

```
[141]: b
```

```
[141]: array([[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11]])
```

```
[142]: b[0]
```

```
[142]: array([0, 1, 2, 3])
```

```
[143]: b[0, :]
```

```
[143]: array([0, 1, 2, 3])
```

The missing indices are considered complete slices: `b[0]` is equivalent to `b[0,:]`

Exercise:

```
b = np.arange(12).reshape(3,4)
```

- Obtain each column in the second and third row of `b`
- Obtain the first three rows and columns of `b`

```
[144]: import numpy as np
```

```
[145]: b = np.arange(12).reshape(3,4)
b
```

```
[145]: array([[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11]])
```

```
[146]: b[1:3,]
```

```
[146]: array([[ 4,  5,  6,  7],
           [ 8,  9, 10, 11]])
```

```
[147]: b[1:3,:]
```

```
[147]: array([[ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
```

```
[148]: b[1:3]
```

```
[148]: array([[ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
```

```
[149]: b[0:3,0:3]
```

```
[149]: array([[ 0,  1,  2],
              [ 4,  5,  6],
              [ 8,  9, 10]])
```

```
[150]: b[:,3,:3]
```

```
[150]: array([[ 0,  1,  2],
              [ 4,  5,  6],
              [ 8,  9, 10]])
```

```
[151]: b
```

```
[151]: array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
```

```
[152]: b
```

```
[152]: array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
```

```
[153]: for i in b:
        print(i)
```

```
[0 1 2 3]
```

```
[4 5 6 7]
```

```
[ 8  9 10 11]
```

Iterating over multidimensional arrays is done with respect to the first axis: row by row

```
[154]: for i in b:
        for j in i:
            print(j)
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
6
7
8
9
10
11
```

More flexible indexing - fancy indexing

- Indexing with Arrays of Indices
- Indexing with Boolean Arrays

```
[155]: a = np.arange(12)**2  # the first 12 square numbers
a
```

```
[155]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81, 100, 121])
```

```
[156]: np.__version__
```

```
[156]: '1.24.3'
```

```
[157]: i = np.array([1, 1, 3, 8, 5])  # an array of indices
i
```

```
[157]: array([1, 1, 3, 8, 5])
```

```
[158]: a[i]  # the elements of `a` at the positions `i`
```

```
[158]: array([ 1,  1,  9, 64, 25])
```

```
[159]: j = np.array([[3, 4], [9, 7]])  # a bidimensional array of indices
j
```

```
[159]: array([[3, 4],
             [9, 7]])
```

```
[160]: a
```

```
[160]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81, 100, 121])
```

```
[161]: a[j]  # the same shape as `j`
```

```
[161]: array([[ 9, 16],
             [81, 49]])
```

What if `a` is multidimensional?

```
[162]: a = a.reshape(4,3)
a
```

```
[162]: array([[ 0,  1,  4],
           [ 9, 16, 25],
           [36, 49, 64],
           [81, 100, 121]])
```

```
[163]: i = np.array([[2, 1], # indices for the first dim of `a`
                    [3, 3]])
```

```
[164]: j = np.array([[0, 1], # indices for the second dim of `a`
                    [1, 2]])
```

```
[165]: a[i,j]
```

```
[165]: array([[36, 16],
           [100, 121]])
```

```
[166]: a
```

```
[166]: array([[ 0,  1,  4],
           [ 9, 16, 25],
           [36, 49, 64],
           [81, 100, 121]])
```

```
[167]: b = a > 14
b
```

```
[167]: array([[False, False, False],
           [False,  True,  True],
           [ True,  True,  True],
           [ True,  True,  True]])
```

```
[168]: a[b] # 1d array with the selected elements
```

```
[168]: array([16, 25, 36, 49, 64, 81, 100, 121])
```

use boolean arrays that have the same shape as the original array

```
[169]: a[b] = 0 # All elements of `a` higher than 14 become 0
a
```

```
[169]: array([[0, 1, 4],
           [9, 0, 0],
           [0, 0, 0],
           [0, 0, 0]])
```

3.4 More on Shape manipulation

```
[170]: a = np.arange(20)
a
```

```
[170]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
          17, 18, 19])
```

```
[171]: a.shape
```

```
[171]: (20,)
```

```
[172]: a.shape = (4,5)
```

```
[173]: a
```

```
[173]: array([[ 0,  1,  2,  3,  4],
          [ 5,  6,  7,  8,  9],
          [10, 11, 12, 13, 14],
          [15, 16, 17, 18, 19]])
```

```
[174]: a.shape = (2,4)
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[174], line 1
----> 1 a.shape = (2,4)

ValueError: cannot reshape array of size 20 into shape (2,4)
```

```
[175]: a.shape = (2,10)
```

```
[176]: a
```

```
[176]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
          [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

```
[177]: a.transpose() # Transpose of the array
```

```
[177]: array([[ 0, 10],
          [ 1, 11],
          [ 2, 12],
          [ 3, 13],
          [ 4, 14],
          [ 5, 15],
          [ 6, 16],
          [ 7, 17],
```

```
[ 8, 18],  
[ 9, 19]])
```

```
[178]: a.T # Transpose of the array
```

```
[178]: array([[ 0, 10],  
             [ 1, 11],  
             [ 2, 12],  
             [ 3, 13],  
             [ 4, 14],  
             [ 5, 15],  
             [ 6, 16],  
             [ 7, 17],  
             [ 8, 18],  
             [ 9, 19]])
```

```
[179]: a.reshape(4,5)
```

```
[179]: array([[ 0,  1,  2,  3,  4],  
             [ 5,  6,  7,  8,  9],  
             [10, 11, 12, 13, 14],  
             [15, 16, 17, 18, 19]])
```

```
[180]: a.reshape(4,-1)
```

```
[180]: array([[ 0,  1,  2,  3,  4],  
             [ 5,  6,  7,  8,  9],  
             [10, 11, 12, 13, 14],  
             [15, 16, 17, 18, 19]])
```

If in a reshaping operation a dimension is given as -1, it is automatically calculated to correspond to the other dimensions.

```
[181]: a
```

```
[181]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],  
             [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

```
[182]: a.reshape(7,-1)
```

```
-----  
ValueError                                Traceback (most recent call last)  
Cell In[182], line 1  
----> 1 a.reshape(7,-1)  
  
ValueError: cannot reshape array of size 20 into shape (7,newaxis)
```

```
[183]: mean_row = a.mean(axis=1)
      mean_row
```

```
[183]: array([ 4.5, 14.5])
```

```
[184]: mean_row.shape
```

```
[184]: (2,)
```

```
[185]: a
```

```
[185]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
             [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

```
[186]: a - mean_row
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[186], line 1
----> 1 a - mean_row

ValueError: operands could not be broadcast together with shapes (2,10) (2,)
```

```
[187]: a.shape
```

```
[187]: (2, 10)
```

3.4.1 Broadcasting for 2-d arrays

How NumPy treats arrays with different shapes during arithmetic operations * One dimension has the same size * The other dimension is of size 1

```
[188]: mean_row = a.mean(axis=1)
      mean_row
```

```
[188]: array([ 4.5, 14.5])
```

```
[189]: mean_row.shape
```

```
[189]: (2,)
```

```
[190]: a
```

```
[190]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
             [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

```
[191]: a - mean_row
```



```

-----
ValueError                                Traceback (most recent call last)
Cell In[191], line 1
----> 1 a - mean_row

ValueError: operands could not be broadcast together with shapes (2,10) (2,)

```

```
[192]: mean_row = mean_row.reshape(2, -1)
mean_row
```

```
[192]: array([[ 4.5],
              [14.5]])
```

```
[193]: mean_row.shape
```

```
[193]: (2, 1)
```

```
[194]: a - mean_row
```

```
[194]: array([[ -4.5, -3.5, -2.5, -1.5, -0.5,  0.5,  1.5,  2.5,  3.5,  4.5],
              [ -4.5, -3.5, -2.5, -1.5, -0.5,  0.5,  1.5,  2.5,  3.5,  4.5]])
```

3.4.2 Concatenating two numpy arrays

```
[195]: a
```

```
[195]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
              [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

```
[196]: a.flatten() # turn the array into 1-d
```

```
[196]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
              17, 18, 19])
```

Stacking arrays together

```
[197]: a
```

```
[197]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
              [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

```
[198]: a.shape
```

```
[198]: (2, 10)
```

```
[199]: b=np.arange(200).reshape(-1,10)
b
```

```
[199]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
              [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
              [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
              [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
              [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
              [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
              [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
              [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
              [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
              [90, 91, 92, 93, 94, 95, 96, 97, 98, 99],
              [100, 101, 102, 103, 104, 105, 106, 107, 108, 109],
              [110, 111, 112, 113, 114, 115, 116, 117, 118, 119],
              [120, 121, 122, 123, 124, 125, 126, 127, 128, 129],
              [130, 131, 132, 133, 134, 135, 136, 137, 138, 139],
              [140, 141, 142, 143, 144, 145, 146, 147, 148, 149],
              [150, 151, 152, 153, 154, 155, 156, 157, 158, 159],
              [160, 161, 162, 163, 164, 165, 166, 167, 168, 169],
              [170, 171, 172, 173, 174, 175, 176, 177, 178, 179],
              [180, 181, 182, 183, 184, 185, 186, 187, 188, 189],
              [190, 191, 192, 193, 194, 195, 196, 197, 198, 199]])
```

```
[200]: b.shape
```

```
[200]: (20, 10)
```

```
[201]: a
```

```
[201]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
              [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

```
[202]: np.vstack((a,b)) #Stack arrays in sequence vertically (row wise): number of
      ↪ columns have to match
```

```
[202]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
              [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
              [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
              [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
              [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
              [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
              [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
              [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
              [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
              [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
              [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
              [90, 91, 92, 93, 94, 95, 96, 97, 98, 99],
              [100, 101, 102, 103, 104, 105, 106, 107, 108, 109],
              [110, 111, 112, 113, 114, 115, 116, 117, 118, 119],
              [120, 121, 122, 123, 124, 125, 126, 127, 128, 129],
```

```
[130, 131, 132, 133, 134, 135, 136, 137, 138, 139],
[140, 141, 142, 143, 144, 145, 146, 147, 148, 149],
[150, 151, 152, 153, 154, 155, 156, 157, 158, 159],
[160, 161, 162, 163, 164, 165, 166, 167, 168, 169],
[170, 171, 172, 173, 174, 175, 176, 177, 178, 179],
[180, 181, 182, 183, 184, 185, 186, 187, 188, 189],
[190, 191, 192, 193, 194, 195, 196, 197, 198, 199]])
```

```
[203]: np.hstack((a,b)) #Stack arrays in sequence horizontally (column wise): : number
      ↪ of rows have to match
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[203], line 1
----> 1 np.hstack((a,b))

File <__array_function__ internals>:200, in hstack(*args, **kwargs)

File ~/opt/anaconda3/lib/python3.9/site-packages/numpy/core/shape_base.py:370,
      ↪ in hstack(tup, dtype, casting)
    368     return _nx.concatenate(arrs, 0, dtype=dtype, casting=casting)
    369 else:
--> 370     return _nx.concatenate(arrs, 1, dtype=dtype, casting=casting)

File <__array_function__ internals>:200, in concatenate(*args, **kwargs)

ValueError: all the input array dimensions except for the concatenation axis
      ↪ must match exactly, but along dimension 0, the array at index 0 has size 2 and
      ↪ the array at index 1 has size 20
```

3.5 Other array functions

```
[204]: a=np.array( [[ 7,2, 1], [4,3, 8] ])
      a
```

```
[204]: array([[7, 2, 1],
             [4, 3, 8]])
```

```
[205]: np.sort?
```

```
[206]: np.sort(a)
```

```
[206]: array([[1, 2, 7],
             [3, 4, 8]])
```

```
[207]: np.sort(a,axis=1)
```

```
[207]: array([[1, 2, 7],  
            [3, 4, 8]])
```

```
[208]: np.sort(a,axis=0)
```

```
[208]: array([[4, 2, 1],  
            [7, 3, 8]])
```

3.6 Further reading

- read [Numpy tutorial](#) to learn more about numpy functionalities