```
import numpy as np
dir(np)
```

Properties:

1. **ndarray:** NumPy's multidimensional array object.
2. **x.size:** Get the number of elements in the array.
3. **shape:** Get the shape of an array.
4. **x.itemsize:** Get the size of each array element in bytes.

Array Creation:

1. **np.zeros():** Create an array filled with zeros.
   Eg: zeros_array = np.zeros((2, 3))

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

2. **np.array():** Create an array from a Python list.
   Eg: python_list = [1, 2, 3, 4, 5]
   array_from_list = np.array(python_list)

3. **np.arange():** Create an array with evenly spaced values within a given range.
   Eg: result = np.arange(0, 11, 2)

```
[ 0  2  4  6  8 10]
```

4. **reshape():** Reshape an array into a specified shape.
   Eg: array = np.arange(12)
   reshaped_array = array.reshape((3, 4))

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

Array Manipulation:

1. **np.nonzero():** Return the indices of non-zero elements in an array.
   Eg: array = np.array([1, 0, 2, 0, 3, 0, 4])
   indices = np.nonzero(array)

```
(array([0, 2, 4, 6]),)
```

2. **np.eye():** Create a 2-D array with ones on the diagonal and zeros elsewhere.
   Eg: identity_matrix = np.eye(3)

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

3. **np.diag():** Extract a diagonal or construct a diagonal array.
   Eg: array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
   diagonal_elements = np.diag(array)

```
[1 5 9]
```

4. **np.fliplr():** Flip array in the left/right direction.
   Eg: array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
   flipped_array = np.fliplr(array)

```
[[3 2 1]
 [6 5 4]
 [9 8 7]]
```

5. **np.tile():** Construct an array by repeating a given array.
   Eg: array = np.array([1, 2, 3])
   tiled_array = np.tile(array, 3)

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

6. **np.linspace():** Create an array of evenly spaced numbers over a specified interval.
   Eg: evenly_spaced_array = np.linspace(0, 1, 5)

```
[0.   0.25 0.5  0.75 1.  ]
```

7. **np.array_split():** Split an array into multiple sub-arrays.
   Eg: array = np.array([1, 2, 3, 4, 5, 6])
   sub_arrays = np.array_split(array, 3)

```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

Random Number Generation:

1. **np.random.choice():** Generate a random sample from a given 1-D array.
   Eg: array = np.array([1, 2, 3, 4, 5])
   random_sample = np.random.choice(array, size=3)

```
[5 4 4]
```

2. **np.random.random():** Return random floats in the half-open interval [0.0, 1.0).
   Eg: random_float = np.random.random(20)

```
[0.99767565 0.68655404 0.14887671 0.79851133 0.73123532 0.66130664
 0.08739076 0.70866555 0.43858592 0.02495141 0.56243524 0.13210976
 0.15145523 0.34204801 0.29842419 0.80593736 0.65777227 0.64750241
 0.78185091 0.38101726]
```

3. **np.random.randint():** Return random integers from low (inclusive) to high (exclusive).
   Eg: x=np.random.randint(-300,300,size=(2,4,3))

```
[[[-229 -178  283]
  [ 186 -287  190]
  [ -64 -273 -276]
  [  77  199  -64]]

 [[ 209  178   17]
  [ -30  166 -201]
  [ 279  -76  -80]
  [  64 -181  100]]]
```

4. **np.random.shuffle():** Modify a sequence in-place by shuffling its contents.
   Eg: array = np.array([1, 2, 3, 4, 5])
   np.random.shuffle(array)

```
[3 1 5 4 2]
```

Statistical Distributions:

1. **np.random.normal():** This function draws random samples from a normal (Gaussian) distribution. It takes parameters for mean, standard deviation, and the size of the output array.

   Syntax: **np.random.normal(loc=0.0, scale=1.0, size=None)**
   - **loc:** Mean of the distribution (default is 0.0).
   - **scale:** Standard deviation of the distribution (default is 1.0).
   - **size:** Output shape (default is None, which returns a single value).

   Ex: samples = np.random.normal(loc=5, scale=2, size=10)

```
[ 5.33187226  3.98393854  4.71503532  4.41652905  7.30509396  4.84893723
  5.28709839  4.72199284  4.96115196 10.25597114]
```

2. **np.random.binomial():** This function draws samples from a binomial distribution. It takes parameters for the number of trials and the probability of success for each trial, as well as the size of the output array.

   Syntax: **np.random.binomial(n, p, size=None)**
   - **n:** Number of trials.
   - **p:** Probability of success for each trial.
   - **size:** Output shape (default is None, which returns a single value).

   Ex: samples = np.random.binomial(n=10, p=0.3, size=10)

```
[3 2 2 4 3 5 2 4 2 5]
```

3. **np.random.exponential():** This function draws random samples from an exponential distribution. It takes a parameter for the scale (inverse of the rate parameter) and the size of the output array.

   Syntax: **np.random.exponential(scale=1.0, size=None)**
   - **scale:** Inverse of the rate parameter (default is 1.0).
   - **size:** Output shape (default is None, which returns a single value).

   Ex: samples = np.random.exponential(scale=2, size=10)

```
[ 4.11122218  5.20276068  0.70652442  2.90373543  1.07234769  1.15969968
  2.80053596  3.02710169  3.23459502 13.10560547]
```

4. **np.random.geometric():** This function draws random samples from a geometric distribution. It takes a parameter for the probability of success of an individual trial and the size of the output array.

   Syntax: **np.random.geometric(p, size=None)**
   - **p:** Probability of success of an individual trial.
   - **size:** Output shape (default is None, which returns a single value).

   Ex: samples = np.random.geometric(p=0.2, size=10)

```
[ 8  1  4  5  7  3  2  3 10  5]
```

5. **np.random.poisson():** This function draws random samples from a Poisson distribution. It takes a parameter for the expected number of occurrences in a fixed interval (lambda) and the size of the output array.

   Syntax: **np.random.poisson(lam, size=None)**
   - **lam:** Expected number of occurrences in a fixed interval.
   - **size:** Output shape (default is None, which returns a single value).

   Ex: samples = np.random.poisson(lam=3, size=10)

```
[4 5 4 1 6 3 1 2 5 3]
```

6. **np.random.logistic():** This function draws random samples from a logistic distribution. It takes parameters for the mean and scale of the distribution, as well as the size of the output array.

   Syntax: **np.random.logistic(loc=0.0, scale=1.0, size=None)**
   - **loc:** Mean of the distribution (default is 0.0).
   - **scale:** Scale parameter (default is 1.0).
   - **size:** Output shape (default is None, which returns a single value).

   Ex: samples = np.random.logistic(loc=10, scale=2, size=10)

```
[12.64821938 11.50995073  8.61826593  8.90729418  9.12565408  7.37267618
  6.91517239  7.30943118  6.09178502 10.33697578]
```

7. **np.random.chisquare():** This function draws random samples from a chi-square distribution. It takes a parameter for the degrees of freedom (shape) and the size of the output array.

   Syntax: **np.random.chisquare(df, size=None)**
   - **df:** Degrees of freedom (shape parameter).
   - **size:** Output shape (default is None, which returns a single value).

   Ex: samples = np.random.chisquare(df=3, size=10)

```
[3.42511656 1.26937284 4.85777681 4.95623844 3.08144306 3.58679757
 3.59543311 4.74034205 0.74425809 1.72085861]
```

8. **np.random.pareto():** This function draws random samples from a Pareto distribution. It takes a parameter for the shape of the distribution and the size of the output array.

   Syntax: **np.random.pareto(a, size=None)**
   - **a:** Shape of the distribution.
   - **size:** Output shape (default is None, which returns a single value).

   Ex: samples = np.random.pareto(a=2, size=10)

```
[ 1.67350836  1.17993429  1.55539854  1.03703279  3.19447777  4.04912364
 21.58432519  1.26693732  2.66283055  1.31560961]
```

9. **dir(np.random):** List available functions and variables in the np.random module.

Array Padding:

The **np.pad()** function pads an array by adding values to its edges. It is commonly used in image processing, convolution operations, and various other numerical computations.

When padding an array, values are added along each dimension of the array. The pad_width parameter determines how many values are added before and after each axis. The mode parameter specifies how the padding values are determined.

Syntax: **numpy.pad(array, pad_width, mode='constant', **kwargs)**

- **array:** The array to be padded.

- **pad_width:** This parameter specifies the number of values padded to the edges of each axis. It can be a scalar, a tuple of scalars, or a list of tuples. Each scalar or tuple represents the number of values padded before and after each axis. If a scalar is provided, it will be applied to all axes equally.

- **mode** (optional): This parameter specifies the padding mode. It can take one of the following values:

    1. **'constant'**: Pads with a constant value.
    2. **'edge'**: Pads with the edge values of the array.
    3. **'linear_ramp'**: Pads with the linear ramp between end value and the constant value.
    4. **'maximum'**: Pads with the maximum value of the array.
    5. **'mean'**: Pads with the mean value of the array.
    6. **'median'**: Pads with the median value of the array.
    7. **'minimum'**: Pads with the minimum value of the array.
    8. **'reflect'**: Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.
    9. **'symmetric'**: Pads with the reflection of the vector mirrored along the edge of the array.
    10. **'wrap'**: Pads with the wrap of the vector along the axis.
    11. **'empty'**: Pads with undefined values.

- ****kwargs** (optional): Additional keyword arguments that may be accepted by certain padding modes.

Ex: array = np.array([[1, 2], [3, 4]])
padded_array1 = np.pad(array, pad_width=1, mode='constant', constant_values=0)

```
[[0, 0, 0, 0],
 [0, 1, 2, 0],
 [0, 3, 4, 0],
 [0, 0, 0, 0]]
```

padded_array2 = np.pad(array, pad_width=2, mode='edge')

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

padded_array3 = np.pad(array, pad_width=1, mode='maximum')

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

Array Indexing and Boolean Operations:

1. **np.where():** Return elements chosen from x or y depending on condition.

The **np.where()** function is used to return the indices of elements in an array that satisfy a given condition. It can be used to extract elements from an array based on a condition, replace elements that meet a condition, or to set values based on a condition.

Syntax: **numpy.where(condition[, x, y])**

- **condition:** This parameter is the condition that the elements of the array should satisfy. It can be a boolean array or an expression that evaluates to a boolean array.
- **x (optional):** If provided, it represents the values in the output array where the condition is True. It can be a scalar, an array, or a callable function.
- **y (optional):** If provided, it represents the values in the output array where the condition is False. It can be a scalar, an array, or a callable function.

Returns:

- If only the condition parameter is provided, np.where() returns a tuple of arrays containing the indices where the condition is True.
- If both x and y parameters are provided, np.where() returns an array with elements from x where the condition is True, and elements from y where the condition is False.

Ex:

```
array = np.array([1, 2, 3, 4, 5])
indices = np.where(array > 3)
```

```
(array([3, 4]),)
```

```
new_array = np.where(array > 3, 0, array)
```

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

Numpy

Array Creation:

1. **np.zeros(shape, dtype=float, order='C'):** Creates an array filled with zeros.

    - shape: The shape of the array, specifying the dimensions (e.g., (3, 4) for a 3x4 matrix).
    - dtype (optional): Data type of the array elements (default is float).
    - order (optional): 'C' for row-major order, 'F' for column-major order (default is 'C').

2. **np.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0):** Creates an array from a Python list or tuple.

    - object: Python list, tuple, or other array-like object.
    - dtype (optional): Data type of the array elements (default is inferred from input).
    - copy (optional): If true (default), the object is copied. If false, a view is created if possible.
    - order (optional): 'C' for row-major order, 'F' for column-major order, 'A' for any (default is 'K').
    - subok (optional): If True, then sub-classes will be passed through (default is False).
    - ndmin (optional): Specifies the minimum number of dimensions the resulting array should have.

3. **np.arange([start,]stop, [step,]dtype=None):** Creates an array with evenly spaced values within a given range.

    - start (optional): Start of the interval (default is 0).
    - stop: End of the interval (not included).
    - step (optional): Spacing between values (default is 1).
    - dtype (optional): Data type of the array elements (default is inferred).

4. **reshape(newshape, order='C'):** Reshapes an array into a specified shape.

    - newshape: The new shape as a tuple of integers.
    - order (optional): 'C' for row-major order, 'F' for column-major order (default is 'C').

Array Manipulation:

1. **np.eye(N, M=None, k=0, dtype=<class 'float'>):** Creates a 2-D array with ones on the diagonal and zeros elsewhere.

    - N: Number of rows.
    - M (optional): Number of columns (default is N).
    - k (optional): Index of the diagonal (default is 0).

2. **np.diag(v, k=0):** Extracts a diagonal or constructs a diagonal array.

- v: If v is a 2-D array, returns the diagonal of v. If v is a 1-D array, constructs a 2-D array with v on the diagonal.
- k (optional): Index of the diagonal (default is 0).

3. **np.tile(A, reps):** Constructs an array by repeating a given array A.

   - A: Input array to be repeated.
   - reps: Number of repetitions of A along each axis.

4. **np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0):** Creates an array of evenly spaced numbers over a specified interval.

   - start: Start of the interval.
   - stop: End of the interval.
   - num (optional): Number of samples to generate (default is 50).
   - endpoint (optional): If True, stop is the last sample; otherwise, it is not included (default is True).
   - retstep (optional): If True, return (samples, step) where step is the spacing between samples.
   - dtype (optional): Data type of the output array (default is inferred).

5. **np.array_split(ary, indices_or_sections, axis=0):** Splits an array into multiple sub-arrays.

   - ary: Array to be split.
   - indices_or_sections: If an integer, it indicates the number of equally sized sub-arrays to create. If a 1-D array of sorted integers, it indicates the indices at which to split.
   - axis (optional): Axis along which to split (default is 0).

Random Number Generation:

1. **np.random.choice(a, size=None, replace=True, p=None):** Generates a random sample from a given 1-D array a.

   - a: Input array.
   - size (optional): Output shape.
   - replace (optional): Whether the sample is with or without replacement (default is True).
   - p (optional): Probabilities associated with each entry in a (default is uniform).

2. **np.random.random(size=None):** Returns random floats in the half-open interval [0.0, 1.0).

   - (optional): Output shape.

3. **np.random.randint(low, high=None, size=None, dtype=int):** Returns random integers from low (inclusive) to high (exclusive).

   - low: Lowest (signed) integer to be drawn from the distribution.
   - high (optional): If provided, one above the largest (signed) integer to be drawn from the distribution (default is None).

- size (optional): Output shape.
- dtype (optional): Data type of the output array (default is int).

4. **np.random.shuffle(x):** Modifies a sequence x in-place by shuffling its contents.

Array Padding:

1. **np.pad(array, pad_width, mode='constant', **kwargs):** Pads an array array.
   - array: Input array.
   - pad_width: Number of values padded to the edges of each axis.
   - mode (optional): Padding mode