Chapter 7. NumPy Arrays: 1D

```
import numpy as np

me = 9.11e-31      # mass of electron
c  = 299792458     # speed of light

u  = 0.1 * c       # particle velocity

gamma = 1 / np.sqrt(1-(u/c)**2)   # gamma factor

KE = (gamma-1) * me * c**2        # relativistic kinetic energy
```

Python for Physicists

Create NumPy Array from Python List

Use the np.array function:

```
A = np.array([10, 20, 30, 40, 50])
```

Copy NumPy arrays

```
B = A.copy()
```

Fetching and Slicing works the same way as Python Lists

```
A[0]      # selects first element (index = 0)
A[1]      # selects 2nd element (index = 1)
A[-1]     # selects last element
A[1:4]    # selects elements 1,2,3
```

Vectorized Operations

Vectorized functions act on all elements of the array

```
A = np.array([3, 6, 9, 12])    # define a NumPy array from a list
A2 = A**2                      # square each element
print("square of A = ",A2)     # display the result
```

```
square of A = [9, 36, 81, 144]
```

```
#####  Python loop:  each element squared "by hand". SLOWER
for i in range(len(A)):
    A[i] = A[i]**2
print("method 2:  A^2 = ",A)
```

Vectorized Operations

Numpy Arrays can be used like variables in an equation if the arrays have the same length

```
x = np.array([1,2,4,8])  # NumPy array (length 4)
y = np.array([3,0,1,1])  # NumPy array (length 4)

z = x*y                  # element-wise product of x and y
print("x*y = ",z)        # result is also length 4
```

```
x*y =  [3 0 4 8]
```

Example

```
theta_deg = np.array([0, 10, 20, 30, 40, 50])   # angles in degrees
theta_rad = np.deg2rad(theta_deg)               # convert the anges
to radians
y_val = np.sin(theta_rad)                       # take the sine of
each angle

for th,y in zip(theta_deg,y_val):               # print results
    print(f"sin({th:2.0f}) = {y:6.4f}")
```

```
sin( 0) = 0.0000
sin(10) = 0.1736
sin(20) = 0.3420
sin(30) = 0.5000
sin(40) = 0.6428
sin(50) = 0.7660
```

NumPy Statistics

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

##### Attributes do not take parentheses
A.size          # number of elements
A.dtype         # data type of the array

##### Methods require parentheses
A.sum()         # sum of the elements
A.prod()        # product of the elements
A.mean()        # mean of the elements
A.std()         # standard deviation
A.var()         # variance
A.min()         # minimum value
A.max()         # maximum value
A.cumsum()      # cumulative sum (result will have same length as A)
A.cumprod()     # cumulative product (result will have same length as A)
```

Creating NumPy Arrays - Linearly Spaced Arrays

Create a linearly spaced array with N elements equally spaced from start to stop inclusive

```
np.linspace(start,stop,N)
```

```
my_array = np.linspace(100,1000,10)
print("my_array is",my_array)
```

```
my_array is [ 100.  200.  300.  400.  500.  600.  700.  800.  900. 1000.]
```

Creating NumPy Arrays - Linearly Spaced Arrays

Create a linearly spaced array with N elements equally spaced from start to stop exclusive with step size = interval

```
np.arange(start,stop,interval)
```

```
my_array = np.arange(0,5.5,0.5)
print("my_array is",my_array)
```

```
my_array is [0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
```

Creating NumPy Arrays - Constant Values

To create a NumPy array with length N and filled with **zeros**, use np.zeros(N)

```
my_array = np.zeros(5)
print("my_array is",my_array)
```

```
my_array is [0. 0. 0. 0. 0. ]
```

To create a NumPy array with length N and filled with **ones**, use np.ones(N)

```
my_array = np.ones(5)
print("my_array is",my_array)
```

```
my_array is [1. 1. 1. 1. 1. ]
```

Creating NumPy Arrays - Constant Values

How would you create a NumPy array of 10 elements filled with 4's?

```
my_array = 4 * np.ones(10)
print("my_array is",my_array)
```

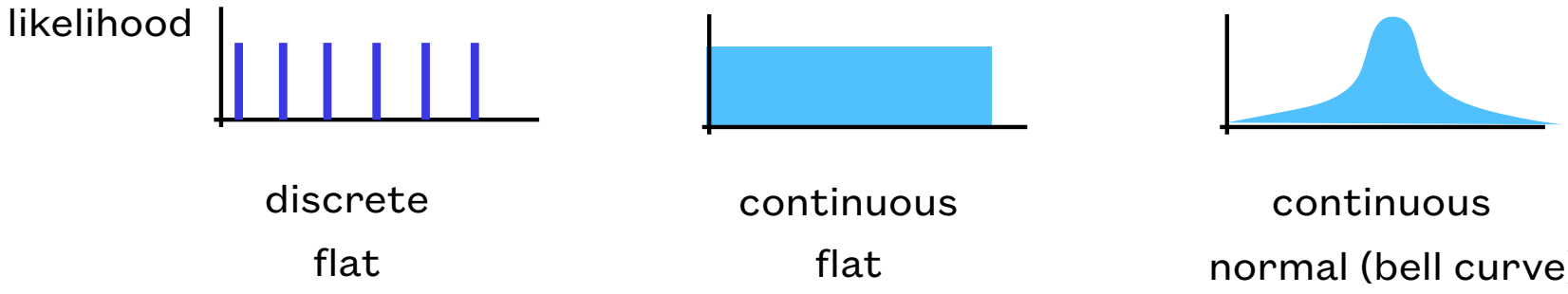
```
my_array is [4. 4. 4. 4. 4. 4. 4. 4. 4. 4. ]
```

Random Numbers

A **random number** is a value generated in such a way that it cannot be predicted in advance and is chosen according to some **probability distribution**.

A probability distribution tells us how likely a given outcome arises on average.

Probability distributions can be classified by their shape and whether they are discrete or continuous.



Generating Random Numbers in NumPy

The NumPy library has its own functions to generate arrays of random numbers.

- In order to use these random number generators, an "instance" of the random number generator object must first be created. To do this we use the command:

```
rng = np.random.default_rng()
```

- Once the `rng` object is created, we can use it to produce a variety of random numbers drawn from different distributions.
- Here are just a few of the many functions available:

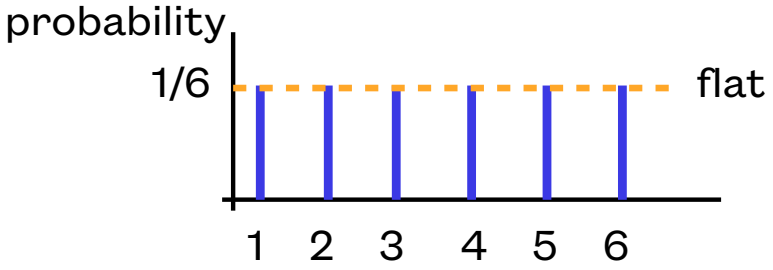
```
rng.integers()      # random integers drawn from flat distribution
rng.choice()        # random elements of a list drawn from flat distribution
rng.uniform()       # random floats drawn from a flat distribution
rng.normal()        # random floats drawn from normal distribution
```

Random Integers drawn from a Flat Probability Distribution

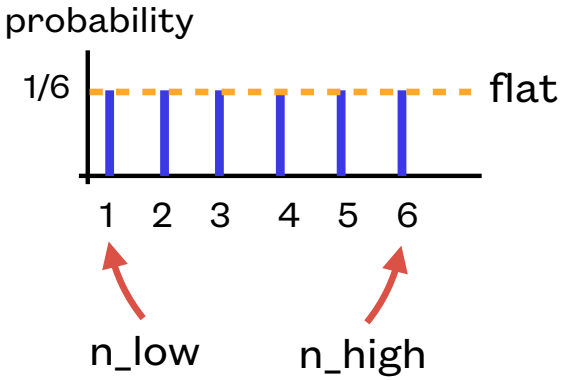
Example: 6-sided die

- Assuming the die is "fair", each side will a 1/6 chance of being rolled, i.e. $P_i = 1/6$.
- Probability distributions that are **flat** have equal chances for every outcome.
- For discrete distributions, the sum of the probabilities of all outcomes must be unity

$$\Sigma_i P_i = 1$$



Random Integers drawn from a Flat Probability Distribution



The command for generating N random integers y_i drawn from a flat probability distribution on $n_{low} \leq y \leq n_{high}$ is:

```
z = rng.integers(n_low, n_high, size=N, endpoint=True)
```

Example function call to produce 10 random rolls of a 6-sided die:

```
y = rng.integers(1, 6, size=10, endpoint=True)
print("dice rolls = ",y)
```

```
dice rolls = [4, 5, 1, 1, 3, 4, 6, 3, 1, 3]
```

Random Integers drawn from a Flat Probability Distribution

The `endpoint=True` option tells Python to include `n_high` in the generated random integers. Without this option, `n_high` is by default an exclusive upper limit, meaning that it is not included

Examples:

```
# Generates 10 random numbers from 1 to 6 inclusive
y = rng.integers(1, 6, size=10, endpoint=True)

# Generates 10 random numbers from 1 to 6 exclusive  (does not include 6)
y = rng.integers(1, 6, size=10)
```

rng.choice() gives more flexibility over rng.integers()

rng.integers() is best if you want to sample a flat distribution of integers.

rng.choice() is best if you need more flexibility, including:

- drawing from a list of items or non-concurrent integers, i.e. [“cat”, “dog”, “mouse”] or [1, 3, 5]
- drawing from a non-flat distribution. rng.choice() allows you to weight each outcome
- ensuring that the random items do not repeat (i.e. no duplicates)

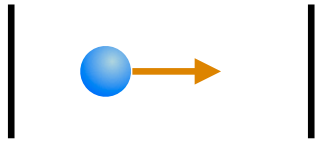
rng.choice() gives more flexibility over rng.integers()

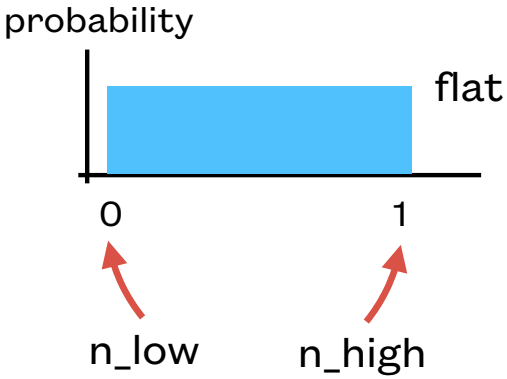
Examples:

```
# pick a random element from a list
rng.choice(["dog", "cat", "mouse"])

# pick a 2 random elements from a list (may get same result twice)
rng.choice(["dog", "cat", "mouse"], size=2)

# pick a 2 unique random elements from a list (no repeats)
rng.choice(["dog", "cat", "mouse"], size=2, replace=False)

# pick a 2 random elements with sampling weights given by p
rng.choice(["dog", "cat", "mouse"], size=2, p=[0.7, 0.2, 0.1])

# pick a 2 unique random numbers from the the range 0-4 (does not include 5)
rng.choice(5, size=2, replace=False)
```

Random Floats drawn from a Flat Probability Distribution

Example: ball with constant speed bouncing between 2 walls





The command for generating a random float y_i drawn from a flat probability distribution on $n_{low} \leq y < n_{high}$ is (note: n_high is not included):

```
z = rng.uniform(n_low, n_high, size=N)
```

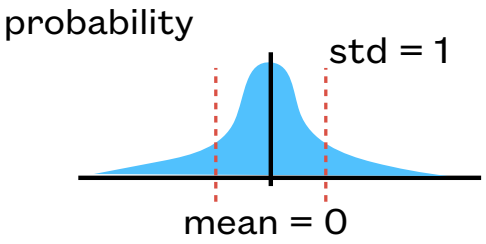
Example function call to produce 4 random values between 0 and 1:

```
y = rng.uniform(0, 1, size=4)
print("random positions = ",y)
```

```
random positions = [0.4586, 0.1994, 0.9443, 0.6753]
```

Random Floats drawn from a Normal Distribution

Normal Distribution



The command for generating N random floats y_i drawn from a Normal probability distribution (i.e. a Bell curve) with mean = 0 and standard deviation = 1 is:

```
z = rng.normal(size=N)
```

Example function call to produce 4 random values:

```
y = rng.normal(size=4)
print("random positions = ",y)

random positions = [-0.342, 0.1554, 1.454, -0.4567]
```