Array-Oriented Programming with NumPy - Part 2

1. `array` Operators
2. Broadcasting
3. Universal Functions (Vectorization)

`array` Operators

The slowness of loops

The speed of computations on `NumPy` `arrays` can range from very fast to very slow. To optimize performance, the recommended approach is to use ***vectorized operations***, which are typically implemented through `NumPy`'s universal functions (`ufuncs`).

The speed of computations on `NumPy` `arrays` can range from very fast to very slow. To optimize performance, the recommended approach is to use ***vectorized operations***, which are typically implemented through `NumPy`'s universal functions (`ufuncs`).

In scenarios involving numerous small operations executed repeatedly, the inherent latency of Python often becomes noticeable. This is particularly the case when looping over arrays to perform operations on each element.

The speed of computations on `NumPy` `arrays` can range from very fast to very slow. To optimize performance, the recommended approach is to use ***vectorized operations***, which are typically implemented through `NumPy`'s universal functions (`ufuncs`).

In scenarios involving numerous small operations executed repeatedly, the inherent latency of Python often becomes noticeable. This is particularly the case when looping over arrays to perform operations on each element.

In [4]:
```
def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0 / values[i]
    return output
```

In [5]:

```
values = np.random.randint(1, 10, 5)
print(values)
compute_reciprocals(values)
```

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

Out[5]:

```
array([0.2       , 0.25      , 0.25      , 0.33333333, 0.2       ])
```

In [5]:
```
values = np.random.randint(1, 10, 5)
print(values)
compute_reciprocals(values)
```

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

Out[5]:
```
array([0.2       , 0.25      , 0.25      , 0.33333333, 0.2       ])
```

But if we measure the execution time of this code for a large input, we see that this operation is very slow:

In [5]:
```
values = np.random.randint(1, 10, 5)
print(values)
compute_reciprocals(values)
```

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

Out[5]:
```
array([0.2       , 0.25      , 0.25      , 0.33333333, 0.2       ])
```

But if we measure the execution time of this code for a large input, we see that this operation is very slow:

In [6]:
```
big_array = np.random.randint(1, 10, 1_000_000)
```

In [5]:
```
values = np.random.randint(1, 10, 5)
print(values)
compute_reciprocals(values)
```

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

Out[5]:
```
array([0.2       , 0.25      , 0.25      , 0.33333333, 0.2       ])
```

But if we measure the execution time of this code for a large input, we see that this operation is very slow:

In [6]:
```
big_array = np.random.randint(1, 10, 1_000_000)
```

In [7]:
```
%%timeit
compute_reciprocals(big_array)
```

```
1.21 s ± 10.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In `NumPy`, ***vectorization*** is the process of performing operations on entire `arrays` of data, as opposed to individual elements. This is accomplished by applying an operation to the entire `array`, instead of looping through each element of the `array` one at a time.

In `NumPy`, ***vectorization*** is the process of performing operations on entire `arrays` of data, as opposed to individual elements. This is accomplished by applying an operation to the entire `array`, instead of looping through each element of the `array` one at a time.

In [8]:
```
print(values)
1.0 / values # The vectorized version of the above code
```

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

Out[8]:
```
array([0.2       , 0.25      , 0.25      , 0.33333333, 0.2       ])
```

In `NumPy`, ***vectorization*** is the process of performing operations on entire `arrays` of data, as opposed to individual elements. This is accomplished by applying an operation to the entire `array`, instead of looping through each element of the `array` one at a time.

In [8]:

```
print(values)
1.0 / values # The vectorized version of the above code
```

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

Out[8]:

```
array([0.2       , 0.25      , 0.25      , 0.33333333, 0.2       ])
```

The above syntax is the vectorized version of the original code and works due to the ***broadcasting***.

Looking at the execution time for our big `array`, we see that it completes orders of magnitude faster than the `Python` loop:

Looking at the execution time for our big `array`, we see that it completes orders of magnitude faster than the `Python` loop:

In [9]:
```
%%timeit
(1.0 / big_array)
```

```
2.09 ms ± 107 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Looking at the execution time for our big `array`, we see that it completes orders of magnitude faster than the `Python` loop:

In [9]:
```
%%timeit
(1.0 / big_array)
```

```
2.09 ms ± 107 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The execution time is much faster since the vectorization operation is done via `ufuncs`, which is a compiled routine.

Element-wise arithmetic

`NumPy` offers numerous operators that allow us to create simple expressions that carry out operations on whole arrays and returns another `array`. Firstly, let's perform **element-wise arithmetic with arrays and a scalar value** by employing arithmetic operators and augmented assignments.

`NumPy` offers numerous operators that allow us to create simple expressions that carry out operations on whole arrays and returns another `array`. Firstly, let's perform **element-wise arithmetic with arrays and a scalar value** by employing arithmetic operators and augmented assignments.

Element-wise operations are applied to each element, so the snippet below doubles every element and cubes every element. Each operation returns a new array containing the result:

`NumPy` offers numerous operators that allow us to create simple expressions that carry out operations on whole arrays and returns another `array`. Firstly, let's perform **element-wise arithmetic with arrays and a scalar value** by employing arithmetic operators and augmented assignments.

Element-wise operations are applied to each element, so the snippet below doubles every element and cubes every element. Each operation returns a new array containing the result:

In [10]:
```
numbers = np.arange(1, 7) # array([1, 2, 3, 4, 5, 6])
numbers * 2
```

Out[10]:
```
array([ 2,  4,  6,  8, 10, 12])
```

`NumPy` offers numerous operators that allow us to create simple expressions that carry out operations on whole arrays and returns another `array`. Firstly, let's perform **element-wise arithmetic with arrays and a scalar value** by employing arithmetic operators and augmented assignments.

Element-wise operations are applied to each element, so the snippet below doubles every element and cubes every element. Each operation returns a new array containing the result:

In [10]:
```
numbers = np.arange(1, 7) # array([1, 2, 3, 4, 5, 6])
numbers * 2
```

Out[10]:
```
array([ 2,  4,  6,  8, 10, 12])
```

In [11]:
```
numbers ** 3
```

Out[11]:
```
array([  1,   8,  27,  64, 125, 216], dtype=int32)
```

Augmented assignments modify every element in the left operand in place!

Augmented assignments modify every element in the left operand in place!

In [12]:
```
numbers += 10
numbers
```

Out[12]:
```
array([11, 12, 13, 14, 15, 16])
```

```
In [13]: display_quiz(path+"list_array2.json", max_width=800)
```

What is printed by the following statements?

```
import numpy as np
lst = [1, 2, 3]
arr = np.array([1, 2, 3])
print(lst + lst)
print(arr + arr)
```

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

```
[2, 4, 6]
[2 4 6]
```

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

```
Error
[2 4 6]
```

Exercise 1: Given the function , estimate the integral of from to using the Riemann sum.

Exercise 1: Given the function , estimate the integral of from to using the Riemann sum.

A Riemann sum is a specific kind of approximation of an integral by a finite sum. It is computed as follows:

Given a function , and a partition of the interval into subintervals, denoted by:

A Riemann sum of this function is constructed as:

Here, is an arbitrary point within each subinterval .

A Riemann sum of this function is constructed as:

Here, is an arbitrary point within each subinterval .

Riemann sums hold significant importance as they allow us to easily approximate a definite integral, represented as:

In [ ]:

```python
# 1. Define the function
def f(x):
    return ___ + ____ + 1
# 2. Generate x values
a, b = ____, ____          # integration limits
n = 1000                 # number of sub-intervals
dx = __________
x = np.linspace(___, ____, n)   # left-endpoint grid

# 3. Compute y values correspind to the left-endpoints
y = f(x)

# 4. Estimate the integral using the Riemann sum
riemann_sum = _____________
print(f"Estimated integral (left Riemann sum, n={n}): {riemann_sum}")

# The exact results
exact_integral = (2**3)/3 + (2**2) + 2  # ∫ (x²+2x+1) dx = x³/3 + x² + x
print(f"Exact integral: {exact_integral}")
```

Broadcasting

Typically, arithmetic operations necessitate two `arrays` of identical size and shape as operands. When one operand is a scalar, `NumPy` carries out the element-wise calculations as though the scalar were an array of the same shape as the other operand, but with the scalar value present in all its elements.

This is referred to as ***broadcasting***. For instance, `numbers * 2` is equivalent to `numbers * [2, 2, 2, 2, 2, 2]`.

Typically, arithmetic operations necessitate two `arrays` of identical size and shape as operands. When one operand is a scalar, `NumPy` carries out the element-wise calculations as though the scalar were an array of the same shape as the other operand, but with the scalar value present in all its elements.

This is referred to as ***broadcasting***. For instance, `numbers * 2` is equivalent to `numbers * [2, 2, 2, 2, 2, 2]`.

Broadcasting can also be applied between `arrays` of different sizes and shapes, enabling concise and powerful manipulations. We will present more examples of broadcasting later in this chapter when we introduce `NumPy`'s universal functions.

Arithmetic Operations Between `arrays`

Arithmetic operations and augmented assignments can be performed between arrays of the same shape. Let's multiply the one-dimensional arrays `numbers` and `numbers2`, each containing six elements:

Arithmetic operations and augmented assignments can be performed between arrays of the same shape. Let's multiply the one-dimensional arrays `numbers` and `numbers2`, each containing six elements:

In [14]:
```
import numpy as np
numbers = np.array([11, 12, 13, 14, 15, 16])
numbers2 = np.linspace(1.1, 6.6, 6)
numbers * numbers2 # array([11, 12, 13, 14, 15, 16]) * array([ 1.1,  2.2,  3..
```

Out[14]:
```
array([ 12.1,  26.4,  42.9,  61.6,  82.5, 105.6])
```

Arithmetic operations and augmented assignments can be performed between arrays of the same shape. Let's multiply the one-dimensional arrays `numbers` and `numbers2`, each containing six elements:

In [14]:
```
import numpy as np
numbers = np.array([11, 12, 13, 14, 15, 16])
numbers2 = np.linspace(1.1, 6.6, 6)
numbers * numbers2 # array([11, 12, 13, 14, 15, 16]) * array([ 1.1,  2.2,  3..
```

Out[14]:
```
array([ 12.1,  26.4,  42.9,  61.6,  82.5, 105.6])
```

The outcome is a new `array` created by multiplying the elements of each operand element-wise — `11 * 1.1, 12 * 2.2, 13 * 3.3`, and so on. Arithmetic operations between `arrays` of integers and floating-point numbers result in an `array` of floating-point numbers due to type conversion .

Let's see another example:

Let's see another example:

In [15]:
```
c = np.ones((3, 3))
c * c
```

Out[15]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Let's see another example:

In [15]:
```
c = np.ones((3, 3))
c * c
```

Out[15]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Note that the above operation is not matrix multiplication. To perform matrix multiplication use the `dot()` method!

Let's see another example:

In [15]:
```
c = np.ones((3, 3))
c * c
```

Out[15]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Note that the above operation is not matrix multiplication. To perform matrix multiplication use the `dot()` method!

In [16]:
```
c.dot(c)
```

Out[16]:
```
array([[3., 3., 3.],
       [3., 3., 3.],
       [3., 3., 3.]])
```

The above operation is the same as using the `@` operator:

The above operation is the same as using the `@` operator:

In [17]:
```
c @ c
```

Out[17]:
```
array([[3., 3., 3.],
       [3., 3., 3.],
       [3., 3., 3.]])
```

We can apply broadcasting to `arrays` with different shape. For instance, consider adding a one-dimensional `array` to a two-dimensional `array` and observe the resulting output:

We can apply broadcasting to `arrays` with different shape. For instance, consider adding a one-dimensional `array` to a two-dimensional `array` and observe the resulting output:

In [18]:
```
a = np.array([0, 1, 2])
M = np.ones((3, 3))
print(a.shape, M.shape) # Note a is not (3,1) or (1,3)
M + a
```

```
(3,) (3, 3)
```

Out[18]:
```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

We can apply broadcasting to `arrays` with different shape. For instance, consider adding a one-dimensional `array` to a two-dimensional `array` and observe the resulting output:

In [18]:
```
a = np.array([0, 1, 2])
M = np.ones((3, 3))
print(a.shape, M.shape) # Note a is not (3,1) or (1,3)
M + a
```

```
(3,) (3, 3)
```

Out[18]:
```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

Here, the one-dimensional array `a` is stretched, or broadcasted, across the second dimension in order to match the shape of `M`.

Rules of Broadcasting

In `NumPy`, broadcasting adheres to a strict set of regulations that govern how two `arrays` interact with one another. These rules are as follows:

1. When the number of dimensions between two `arrays` differs, the `array` with fewer dimensions is padded with ones on its leading (left) side to match the number of dimensions of the other `array`.
2. If the shape of the two `arrays` doesn't match in any dimension, the `array` with a shape of 1 in that dimension is expanded to match the shape of the other `array`.
3. If the sizes of the `arrays` conflict in any dimension and neither is equal to 1, an error is raised.

Now let's take a look at an example where both `arrays` need to be broadcast:

Now let's take a look at an example where both `arrays` need to be broadcast:

In [19]:
```
a = np.arange(0, 40, 10).reshape(4,1)
b = np.arange(3)
print(a.shape, b.shape)
a, b
```

```
(4, 1) (3,)
```

Out[19]:
```
(array([[ 0],
        [10],
        [20],
        [30]]),
 array([0, 1, 2]))
```

Now let's take a look at an example where both `arrays` need to be broadcast:

In [19]:
```
a = np.arange(0, 40, 10).reshape(4,1)
b = np.arange(3)
print(a.shape, b.shape)
a, b
```

```
(4, 1) (3,)
```

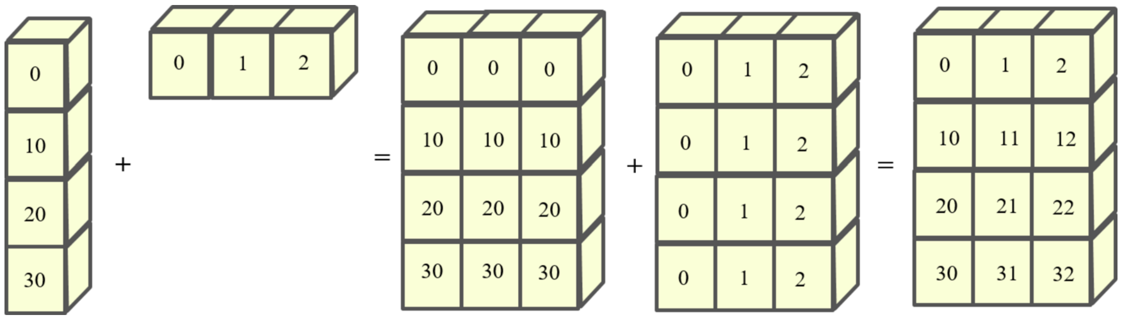
Out[19]:
```
(array([[ 0],
        [10],
        [20],
        [30]]),
 array([0, 1, 2]))
```

In [20]:
```
a + b
```

Out[20]:
```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

This entire process can be depicted visually as follows:

This entire process can be depicted visually as follows:



Next, let's look at an example in which the two `arrays` are incompatible!

Next, let's look at an example in which the two `arrays` are incompatible!

In [21]:
```
M = np.ones((3, 2))
a = np.arange(3)

M.shape, a.shape
```

Out[21]:
```
((3, 2), (3,))
```

Next, let's look at an example in which the two `arrays` are incompatible!

In [21]:
```
M = np.ones((3, 2))
a = np.arange(3)

M.shape, a.shape
```

Out[21]:
```
((3, 2), (3,))
```

In [22]:
```
M + a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_39884\3374645918.py in <module>
----> 1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

```
In [23]: display_quiz(path+"broadcasting.json", max_width=800)
```

What is printed by the following statements?

```
import numpy as np

a = np.array([[1], [2], [3]])   # shape (3,1)
b = np.array([10, 20, 30])      # shape (3,)
print(a + b)
```

```
[[11 11 11]
 [22 22 22]
 [33 33 3
3]]
```

```
None of the abov
e
```

```
ValueError: operands could not be broadcast tog
ether
```

```
[[11 21 31]
 [12 22 32]
 [13 23 3
3]]
```

Exercise 2: Suppose we are dealing with a spreadsheet that records the grade of students. The grade contains the homework, midterm and finals as follows:

Name	HW1	HW2	HW3	HW4	Midterm	Final
Alice	90	80	70	100	90	95
Bob	80	90	100	70	85	80
Charlie	70	100	90	80	95	90
David	60	70	80	90	85	100
Eve	50	60	70	80	75	90

We would like to calculate the semester score of each student by the following rules:

1. The weight of each score is 0.2 (the summation of four homework accounts for 20% of the total scores and each homework has the same weight), 0.4 and 0.4 for HW, Midterm and Final, respectively.

2. We adjust each student's score so that the top performer in the class gets a score of 100 by adding the same constant score to each student's score.

We use a 2D array to model the grades so that each row corresponds to a student's score. Use the following template to complete the task:

```
grades = np.array([[90, 80, 70, 100, 90, 95],
                   [80, 90, 100, 70, 85, 80],
                   [70, 100, 90, 80, 95, 90],
                   [60, 70, 80, 90, 85, 100],
                   [50, 60, 70, 80, 75, 90]])

weights = np.array([])
scores
```

In [ ]:

```
# input data
grades = np.array([[90, 80, 70, 100, 90, 95],
                   [80, 90, 100, 70, 85, 80],
                   [70, 100, 90, 80, 95, 90],
                   [60, 70, 80, 90, 85, 100],
                   [50, 60, 70, 80, 75, 90]])

# weights of HW, Midterm, Final
weights = np.array([])

# calculate weighted score
scores
```

Universal Functions (Vectorization)

Now we will delve into how `NumPy` perform element-wise operations on `arrays` without using the `for` loop: `NumPy` provides most operators/functions as standalone ***universal functions*** (`ufuncs`) that perform various operations element-wise, meaning that they apply the same operation to each element in an `array`.

Now we will delve into how `NumPy` perform element-wise operations on `arrays` without using the `for` loop: `NumPy` provides most operators/functions as standalone ***universal functions*** (`ufuncs`) that perform various operations element-wise, meaning that they apply the same operation to each element in an `array`.

These functions operate on one or two `array`-like arguments and are utilized to perform tasks. Some of these functions are automatically invoked when operators like `+` and `*` are used with `arrays`. Each `ufunc` generates a new `array` that contains the results of the operation.

`NumPy` offers a practical interface that directly access these statically typed and compiled routines. These operations are called ***vectorized operations***. Vectorization is achieved using `array` operations, such as addition, subtraction, multiplication, and division. In addition, it can also be achieved by using other `ufunc`.

These vectorized methods are intended to move the loop to the compiled layer that underpins `NumPy`, leading to considerably quicker execution.

Exploring `NumPy`'s `Ufuncs`

Let's add two `arrays` with the same shape, using the `add()` universal function:

Let's add two `arrays` with the same shape, using the `add()` universal function:

In [24]:
```
import numpy as np
numbers = np.array([11, 12, 13, 14, 15, 16])
numbers2 = np.arange(10, 70, 10)  # array([10, 20, 30, 40, 50, 60])
np.add(numbers, numbers2)        # equivalent to numbers + numbers2
```

Out[24]:
```
array([21, 32, 43, 54, 65, 76])
```

Broadcasting with Universal Functions

Let's use the `multiply()` universal function to multiply every element of `numbers2` by the scalar value 5:

Let's use the `multiply()` universal function to multiply every element of `numbers2` by the scalar value 5:

```
In [25]: np.multiply(numbers2, 5) # equivalent to numbers2 * 5
```

```
Out[25]: array([ 50, 100, 150, 200, 250, 300])
```

Let's reshape `numbers2` into a 2-by-3 array, then multiply its values by a one-dimensional `array` of three elements:

Let's reshape `numbers2` into a 2-by-3 array, then multiply its values by a one-dimensional `array` of three elements:

In [26]:
```
numbers3 = numbers2.reshape(2, 3)
numbers4 = np.array([2, 4, 6])
numbers3, numbers4
```

Out[26]:
```
(array([[10, 20, 30],
        [40, 50, 60]]),
 array([2, 4, 6]))
```

Let's reshape `numbers2` into a 2-by-3 array, then multiply its values by a one-dimensional `array` of three elements:

In [26]:
```
numbers3 = numbers2.reshape(2, 3)
numbers4 = np.array([2, 4, 6])
numbers3, numbers4
```

Out[26]:
```
(array([[10, 20, 30],
        [40, 50, 60]]),
 array([2, 4, 6]))
```

In [27]:
```
np.multiply(numbers3, numbers4) # Equivalent to numbers3 * numbers4
```

Out[27]:
```
array([[ 20,  80, 180],
       [ 80, 200, 360]])
```

There are other special mathematical `ufunc`. Let's create an array and the values using the `sin()` universal function:

There are other special mathematical `ufunc`. Let's create an array and the values using the `sin()` universal function:

In [28]:
```
numbers = np.array([1, 4, 9, 16, 25, 36])
np.sin(numbers)
```

Out[28]:
```
array([ 0.84147098, -0.7568025 ,  0.41211849, -0.28790332, -0.1323517
5,
       -0.99177885])
```

Vectorization and `ufunc` functions are closely associated with broadcasting in `NumPy`. By combining vectorization, `ufunc` functions, and broadcasting, we can effectively execute complex arithmetic operations on `NumPy` `arrays`.

However, it's important to mention that vectorization can be achieved through methods other than just using ufuncs.

Create our own vectorizing functions

The vectorized operation are often more concise, and it is thus advisable to avoid element-wise looping over vectors and matrices and instead employ vectorized algorithms.

The first step in converting a scalar algorithm to a vectorized algorithm involves verifying that the functions we create can function with vector inputs:

The vectorized operation are often more concise, and it is thus advisable to avoid element-wise looping over vectors and matrices and instead employ vectorized algorithms.

The first step in converting a scalar algorithm to a vectorized algorithm involves verifying that the functions we create can function with vector inputs:

In [29]:
```
def Theta(x, th):
    """
    Scalar implemenation of a variant of Heaviside step function.
    """
    if x >= th:
        return 1
    else:
        return 0
```

We can achieve this using `np.vectorize()` function:

We can achieve this using `np.vectorize()` function:

In [30]:
```
Theta_vec = np.vectorize(Theta)
Theta_vec(np.array([-3,-2,-1,0,1,2,3]), 1)
```

Out[30]:
```
array([0, 0, 0, 0, 1, 1, 1])
```

In [31]:

```
display_quiz(path+"universal.json", max_width=800)
```

Which of the following are NumPy universal functions (ufuncs)? (Select all that apply)

np.add	np.sqrt
math.sqrt	np.dot

Exrercise 3: Compare the performance between `for` loop and `NumPy` vectorization in calculating the Wallis formula: $2 \times \prod_{i=1}^{500}(\frac{2i}{2i-1} \times \frac{2i}{2i+1})$. Be sure to check the results from the two approaches are the same and close enough to the true value of π. Finally, report the speedup factor of the `NumPy` vectorization.

Hint: Use `%%timeit` to measure the performance of the code. In addition, look up the official documentation or use copilot to find the `NumPy` function to calculate the product of an array using vectorization.

In [ ]:
```
def wallis_loop(n=N_TERMS):
    product = 1.0
    for i in range(1, n + 1):
        product *= (2*i)/(2*i - 1)
        product *= (2*i)/(2*i + 1)
    return 2 * product
# Your code here
def wallis_vec(n=N_TERMS):
    i = np.arange(____, ____, dtype=float) # Create a 1-D NumPy array [1, 2, .
    terms = _________    # Compute each factor (2i)/(2i-1)*(2i)/(2i+1) element
    return 2 * np._____(terms)    # Multiply all factors together and scale by
```

In [ ]:

```
approx = wallis_vec()
print(f"Wallis approximation : {approx:.12f}")
print(f"π                    : {np.pi:.12f}")
print(f"absolute error       : {abs(approx - np.pi):.6e}")
assert np.allclose(approx, wallis_loop()), "implementations disagree!"
```

In [ ]:

```
approx = wallis_vec()
print(f"Wallis approximation : {approx:.12f}")
print(f"π                    : {np.pi:.12f}")
print(f"absolute error       : {abs(approx - np.pi):.6e}")
assert np.allclose(approx, wallis_loop()), "implementations disagree!"
```

In [ ]:

```
%%timeit -o
wallis_loop()
```

In [ ]:

```
approx = wallis_vec()
print(f"Wallis approximation : {approx:.12f}")
print(f"π                    : {np.pi:.12f}")
print(f"absolute error       : {abs(approx - np.pi):.6e}")
assert np.allclose(approx, wallis_loop()), "implementations disagree!"
```

In [ ]:

```
%%timeit -o
wallis_loop()
```

In [ ]:

```
%%timeit -o
wallis_vec()
```

In summary:

To make the code faster using `NumPy`

- Use views instead of copies whenever possible
- Broadcasting: Use broadcasting to do operations on `arrays`
- Vectorizing `for` loops: Find tricks to avoid `for` loops using `NumPy` `arrays`.
- In place operations: `a *= 3` instead of `a = 3*a`

The comparisons between `list` and `array` are summarized as follows:

Python objects:

- `Python` `lists` are very general. They can contain any kind of object and are dynamically typed
- However, they do not support mathematical functions such as matrix multiplications. Implementing such functions for `Python` `lists` would not be very efficient because of the dynamic typing

NumPy provides:

- `Numpy` `arrays` are **statically typed** and **have the same data type**. The type of the elements is determined when the `array` is created
- Because of this static typing, `NumPy` can utilize fast implementation of mathematical functions using a compiled language (NumPy uses C and Fortran). This contributes to their computational and memory efficiency.
- For scientific computing tasks where efficiency and mathematical operations are key, it is generally recommended to use `NumPy` arrays to model and manipulate the data.

In [39]:

```
from jupytercards import display_flashcards
fpath= "flashcards/"
display_flashcards(fpath + 'ch9-2.json')
```

Vectorization

Next

>