In [1]:
```
import pybryt
from lecture import pybryt_reference
```

Introduction to Python

Lecture 2

Learning objectives:

At the end of this lecture, you will be able to:

- Modify elements in a `list`.
- Iterate through different combinations of lists.
- Use a `tuple` to store data elements and understand how it differs from a `list`.
- Explain the difference between locally-scoped and globally-scoped variables.
- Use an `if`-statement to execute some code blocks conditionally.
- Perform computations using Numerical Python (*NumPy*).
- Handle multidimensional arrays.

Changing elements in a list

Let's say we want to add 2 to all the numbers in a list `v`. To do that, we have to use an index to access and modify its elements:

In [2]:
```
v = [-1, 1, 10]
print(f"List before modification: {v=}")

v[1] = 4  # assign 4 to the 2nd element (index 1) in v
print(f"List after modification: {v=}")
```

```
List before modification: v=[-1, 1, 10]
List after modification: v=[-1, 4, 10]
```

Please note how we used `{v=}` in f-string to add `v=` in front of the list.

Now, to add 2 to all values we need a `for` loop over indices:

In [3]:
```python
v = [-1, 1, 10]
for i in range(len(v)):
    v[i] = v[i] + 2
print(v)
```

```
[1, 3, 12]
```

Note that this time we iterate over the indices of the list elements

```python
for i in range(len(v)):
    ...
```

instead of iterating over the values of elements in the list

```python
for e in v:
    ...
```

`enumerate` built-in

As we have seen previouly, we often need to use both the value of an element in a sequence and its index. Python provides a convenience built-in function `enumerate` to make this syntax clearer:

In [4]:
```python
v = [-1, 1, 10]
for index, value in enumerate(v):
    v[index] = value + 2

print(v)
```

```
[1, 3, 12]
```

Exercise 2.1: Create a list and modify it.

Write a Python function `mult(vector, n)` which takes list `vector` and number `n` as input arguments. The function should multiply each element in the list by `n` using a `for` loop and return the resulting list.

In [5]:
```
# Uncomment and modify the following lines. Do not change any variable names for testing purposes.

def mult(vector, n):
    result = []
    for i in range(len(vector)):
        vector[i] = vector[i] * n
        result.append(vector[i])
    return result

vector = [2.1, 99.9, -10, 2]
n = 3
print(mult(vector, n))
```

```
[6.300000000000001, 299.70000000000005, -30, 6]
```

In [6]:
```
with pybryt.check(pybryt_reference(2, 1)):
    mult([2.1, 99.9, -10, 2], 3)
```

```
REFERENCE: exercise-2_1
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! You are iterating through list indices.
  - SUCCESS: You are multiplying each element by n. Well done!
  - SUCCESS: Wow! Your function returns the correct result.
```

In [7]:
```
import numbers
import numpy as np

res = mult([1, 1.1, 1.11], 4)
assert np.allclose(res, [4, 4.4, 4.44])
assert isinstance(res, list)
assert all([isinstance(i, numbers.Real) for i in res])
assert len(res) == 3

### BEGIN HIDDEN TESTS
assert callable(mult)
```

```
assert mult([], 5) == []
### END HIDDEN TESTS
```

Traversing multiple lists simultaneously: `zip(list1, list2, ...)`

Let us consider how we can loop over elements in both `Cdegrees` and `Fdegrees` at the same time. One approach would be to use list indices:

In [8]:
```
# First, we have to recreate the data from lecture 1.
Cdegrees = [deg for deg in range(-20, 41, 5)]
Fdegrees = [(9/5)*deg + 32 for deg in Cdegrees]

for i in range(len(Cdegrees)):
    print(Cdegrees[i], Fdegrees[i])
```

```
-20 -4.0
-15 5.0
-10 14.0
-5 23.0
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
40 104.0
```

An alternative construct, regarded as more ”Pythonic”, uses the `zip` built-in function:

In [9]:
```
for C, F in zip(Cdegrees, Fdegrees):
    print(C, F)
```

```
-20 -4.0
-15 5.0
-10 14.0
-5 23.0
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
40 104.0
```

Using `zip`, we can also traverse three or more lists simultaneously:

In [10]:
```
l1 = [3, 6, 1]
l2 = [1, 1, 0]
l3 = [9, 3, 2]

for e1, e2, e3 in zip(l1, l2, l3):
    print(f"{e1 = }, {e2 = }, {e3 = }")
```

```
e1 = 3, e2 = 1, e3 = 9
e1 = 6, e2 = 1, e3 = 3
e1 = 1, e2 = 0, e3 = 2
```

If the lists are of unequal length, then the loop stops when we reach the end of the shortest list. Experiment with this:

In [11]:
```
l1 = [3, 6, 1, 4, 6]  # len(l1) == 5
l2 = [1, 1, 0, 7]     # len(l2) == 4
l3 = [9, 3, 2, 0, 9]  # len(l3) == 5

for e1, e2, e3 in zip(l1, l2, l3):
    print(f"{e1 = }, {e2 = }, {e3 = }")
```

```
e1 = 3, e2 = 1, e3 = 9
e1 = 6, e2 = 1, e3 = 3
e1 = 1, e2 = 0, e3 = 2
e1 = 4, e2 = 7, e3 = 0
```

Nested lists: list of lists

A `list` can contain **any** object as its element, including another `list`. To illustrate this, consider storing the conversion table as a single Python list rather than two separate lists:

In [12]:
```
Cdegrees = [C for C in range(-20, 41, 5)]
Fdegrees = [(9/5)*C + 32 for C in Cdegrees]
table1 = [Cdegrees, Fdegrees]  # List of two lists

print(f"{table1 = }")
print(f"{table1[0] = }")  # access the first element of list table1 - Cdegrees list
print(f"{table1[1] = }")  # access the second element of list table1 - Fdegrees list
print(f"{table1[1][3] = }")  # access 4th element in the 2nd list
```

```
table1 = [[-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40], [-4.0, 5.0, 14.0, 23.0, 32.0, 41.0, 50.0, 59.0, 68.0, 77.0, 8
6.0, 95.0, 104.0]]
table1[0] = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
table1[1] = [-4.0, 5.0, 14.0, 23.0, 32.0, 41.0, 50.0, 59.0, 68.0, 77.0, 86.0, 95.0, 104.0]
table1[1][3] = 23.0
```

This gives us a table with two rows. How do we create a table of columns instead:

In [13]:
```
table2 = []
for C, F in zip(Cdegrees, Fdegrees):
    row = [C, F]
    table2.append(row)

print(table2)
```

```
[[-20, -4.0], [-15, 5.0], [-10, 14.0], [-5, 23.0], [0, 32.0], [5, 41.0], [10, 50.0], [15, 59.0], [20, 68.0], [25, 77.0], [30, 8
6.0], [35, 95.0], [40, 104.0]]
```

We can also use list comprehension to do this more elegantly:

In [14]:
```
table2 = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
print(table2)
```

```
[[-20, -4.0], [-15, 5.0], [-10, 14.0], [-5, 23.0], [0, 32.0], [5, 41.0], [10, 50.0], [15, 59.0], [20, 68.0], [25, 77.0], [30, 8
6.0], [35, 95.0], [40, 104.0]]
```

And we can loop through this list as before:

In [15]:
```
for C, F in table2:
    print(C, F)
```

```
-20 -4.0
-15 5.0
-10 14.0
-5 23.0
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
40 104.0
```

Since elements of `table2` are length-2 lists, in each iteration, we *unpack* each of the length-2 elements to `C` and `F`.

Tuples: lists that cannot be changed

Tuples are **constant** lists, i.e. we can use them in much the same way as lists except we cannot modify them. They are an example of an **immutable** type.

In [16]:
```
t = (2, 4, 6, "temp.pdf")  # Define a tuple.
t = 2, 4, 6, "temp.pdf"    # Can skip parenthesis as it is assumed in this context.
```

Let us see what happens when we try to modify the tuple like we did with a list:

```
t[1] = -1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-3-593c03edf054> in <module>()
----> 1 t[1] = -1

TypeError: 'tuple' object does not support item assignment
```

```
t.append(0)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-19-78592bf72d62> in <module>()
----> 1 t.append(0)

AttributeError: 'tuple' object has no attribute 'append'
```

```
del t[1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-0193a527a912> in <module>()
----> 1 del t[1]

TypeError: 'tuple' object doesn't support item deletion
```

However, we can use the tuple to compose a new tuple:

In [17]:
```
t = t + (-1.0, -2.0)
print(t)
```

```
(2, 4, 6, 'temp.pdf', -1.0, -2.0)
```

So, why would we use tuples when lists have more functionality?

- Tuples are constant and thus *protected against accidental changes*.
- Tuples are *faster* than lists.
- Tuples are *widely used* in Python software (so you need to know about tuples to understand other people's code!)
- Tuples (but not lists) are hashable and can be used as *keys in dictionaries* (more about dictionaries later).

WARNING: Tuples are actually not always immutable. If a tuple contains mutable elements, e.g. `list`, it is possible to change the tuple. Let us have a look at this example:

In [18]:
```
t = (1, 2, [3, 4])
t[2].append(5)  # we are appending 5 to the list which the tuple holds a reference of
print(t)
```

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

Therefore, to ensure the tuple is immutable, it is necessary for it to contain only immutable elements. The best way to check if the tuple is actually mutable is to use the `hashable` function.

In [19]:
```
t = (1, 2, [3, 4])  # tuple contains a list which is a mutable type
try:
    hash(t)
    print(f"Tuple {t = } is immutable.")
except TypeError:
    print(f"Tuple {t = } is mutable.")
```

```
Tuple t = (1, 2, [3, 4]) is mutable.
```

In [20]:
```
t = (1, 2, "abc", (5, 6))  # tuple contains only immutable types
try:
    hash(t)
    print(f"Tuple {t = } is immutable.")
except TypeError:
    print(f"Tuple {t = } is mutable.")
```

```
Tuple t = (1, 2, 'abc', (5, 6)) is immutable.
```

What does it then mean that some types are mutable and some are not? We will talk about this in lecture 4.

Exercise 2.2: Make a table (a list of lists) of function values

- Write a loop that evaluates the expression $y(t) = v_0 t – {1\over2}gt^2$ for 11 evenly spaced t values ranging from 0, to $2v_0/g$ (remember that dividing a range into n intervals results in $n+1$ values). You can assume that $v_0 = 1\,$ms^{-1} and $g = 9.81\,$ms^{-2}.

- Store the time values and displacement (y) values as a nested list, i.e.

```
tlist = [t0, t1, t2, ...]
ylist = [y0, y1, y2, ...]
displacement = [tlist, ylist]
```

- Use the variable names `tlist`, `ylist` and `displacement` as illustrated in the above example for testing purposes.

In [21]:
```
# Uncomment and modify the following lines. Do not change variable names for testing purposes.
v0 = 1.0
g = 9.81
n = 10

# Step size for time
t_max = 2 * v0 / g
dt = t_max / n

# Generate lists
tlist = []
ylist = []

for i in range(n+1):
    t = i * dt
    y = v0 * t - 0.5 * g * t**2
    tlist.append(t)
    ylist.append(y)

# Nested list
displacement = [tlist, ylist]

# For checking
print("tlist =", tlist)
print("ylist =", ylist)
print("displacement =", displacement)
```

```
tlist = [0.0, 0.02038735983690112, 0.04077471967380224, 0.061162079510703356, 0.08154943934760447, 0.1019367991845056, 0.122324
15902140671, 0.14271151885830782, 0.16309887869520895, 0.18348623853211007, 0.2038735983690112]
ylist = [0.0, 0.018348623853211007, 0.03261977573904179, 0.04281345565749235, 0.04892966360856269, 0.0509683995922528, 0.048929
6636085627, 0.04281345565749235, 0.03261977573904182, 0.018348623853211038, 0.0]
displacement = [[0.0, 0.02038735983690112, 0.04077471967380224, 0.061162079510703356, 0.08154943934760447, 0.1019367991845056,
0.12232415902140671, 0.14271151885830782, 0.16309887869520895, 0.18348623853211007, 0.2038735983690112], [0.0, 0.01834862385321
1007, 0.03261977573904179, 0.04281345565749235, 0.04892966360856269, 0.0509683995922528, 0.0489296636085627, 0.0428134556574923
5, 0.03261977573904182, 0.018348623853211038, 0.0]]
```

In [22]:
```python
with pybryt.check(pybryt_reference(2, 2)):
    tlist, ylist, displacement
```

```
REFERENCE: exercise-2_2
SATISFIED: True
MESSAGES:
  - SUCCESS: You generated tlist correctly. Great!
  - SUCCESS: Your ylist is correct. Well done!
```

In [23]:
```python
import numbers
import numpy as np

assert isinstance(tlist, list)
assert isinstance(ylist, list)
assert isinstance(displacement, list)

assert np.isclose(tlist[0], 0)
assert np.isclose(tlist[-1], 0.2038735983690112)
assert np.isclose(ylist[0], 0)
assert np.isclose(ylist[-1], 0)

assert len(tlist) == len(ylist) == 11


### BEGIN HIDDEN TESTS
assert all([isinstance(i, numbers.Real) for i in tlist])
assert all([isinstance(i, numbers.Real) for i in ylist])

assert all([isinstance(i, list) for i in displacement])
assert all([isinstance(i, numbers.Real) for i in displacement[0]])
assert all([isinstance(i, numbers.Real) for i in displacement[0]])
```

```
assert np.allclose(displacement[0], tlist)
assert np.allclose(displacement[1], ylist)
### END HIDDEN TESTS
```

The `if` construct

Let us consider we need to program the following function: $$ f(x)= \begin{cases} \sin(x),& \text{if } 0 \leq x \leq \pi\\ 0, & \text{otherwise} \end{cases} $$ To do this, we need the `if` construct:

In [24]:
```
from math import sin, pi


def f(x):
    if 0 <= x <= pi:
        return sin(x)
    else:
        return 0


print(f"{f(-pi/2) = }")
print(f"{f(pi/2) = }")
print(f"{f(3*pi/2) = }")
```

```
f(-pi/2) = 0
f(pi/2) = 1.0
f(3*pi/2) = 0
```

Please note the indentations we used to define which statements belong to which condition. Sometimes, it is clearer to write this as a conditional expression:

In [25]:
```
def f(x):
    return sin(x) if 0 <= x <= pi else 0


print("f(-pi/2) =", f(-pi/2))
print("f(pi/2) =", f(pi/2))
print("f(3*pi/2) =", f(3*pi/2))
```

```
f(-pi/2) = 0
f(pi/2) = 1.0
f(3*pi/2) = 0
```

The `else` block can be skipped if there are no statements to be executed when `False`. In general, we can put together multiple conditions. Only the first condition that is `True` is executed.

```
if condition1:
    <block of statements, executed if condition1 is True>
elif condition2:
    <block of statements, executed if condition1 is False and condition2 is True>
elif condition3:
    <block of statements, executed if conditions 1 and 2 are False and condition3 is True>
else:
    <block of statements, executed if conditions 1, 2, and 3 are False>

<next statement of the program>
```

Exercise 2.3: Express a step (Heaviside) function as a Python function

The following "step" function is known as the Heaviside function and it is widely used in mathematics: $$ H(x)= \begin{cases} 0, & \text{if}\; x < 0\\ 1, & \text{if}\; x \ge 0. \end{cases} $$ Write a Python function `heaviside(x)` that computes $H(x)$.

In [26]:

```python
# Uncomment and modify the following lines. Do not change variable names for testing purposes.

def heaviside(x):
    if x < 0:
        return 0
    else:
        return 1
print("H(-1000) = ", heaviside(-1000))
print("H(1000) = ", heaviside(1000))
print("H(0) = ", heaviside(0))
```

```
H(-1000) =  0
H(1000) =  1
H(0) =  1
```

In [27]:
```
with pybryt.check(pybryt_reference(2, 3)):
    heaviside(-1000), heaviside(1000), heaviside(0)
```

```
REFERENCE: exercise-2_3
SATISFIED: True
MESSAGES:
  - SUCCESS: Your function returns correct value for negative x.
  - SUCCESS: Your function returns correct value for positive x. Well done!
  - SUCCESS: Amazing! Your function returns correct value for x=0.
```

In [28]:
```
import numbers

assert heaviside(-5.1) == 0
assert heaviside(5.1) == 1
assert heaviside(0) == 1

### BEGIN HIDDEN TESTS
assert callable(heaviside)
assert isinstance(heaviside(6), numbers.Real)
### END HIDDEN TESTS
```

Exercise 2.4: Implement the factorial function

The factorial of n, written as $n!$, is defined as

$$n! = n(n - 1)(n - 2) \cdot \ldots \cdot 2 \cdot 1,$$

with the special cases

$$1! = 1, 0! = 1.$$

For example, $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$, and $2! = 2 \cdot 1 = 2$.

Implement your own factorial function to calculate $n!$. Return 1 immediately if n is 1 or 0; otherwise use a loop to compute $n!$. You can use Python's own math.factorial(x) to check your code.

In [29]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
import math
def my_factorial(n):
    if n == 1 or n == 0:
        return 1
    else:
        result = 1
        for i in range(2, n+1):
            result *= i
        return result
print("my_factorial(10) = ", my_factorial(10))
print("check with math.factorial(10) =", math.factorial(10))
```

```
my_factorial(10) =  3628800
check with math.factorial(10) = 3628800
```

In [30]:
```
with pybryt.check(pybryt_reference(2, 4)):
    my_factorial(10)
```

```
REFERENCE: exercise-2_4
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! You are multiplying values correctly.
  - SUCCESS: Your loop iterates over the correct values.
  - SUCCESS: Your function computes factorial correctly. Well done!
```

In [31]:
```
import numbers

assert my_factorial(0) == 1
assert my_factorial(1) == 1
assert my_factorial(2) == 2
assert my_factorial(5) == 120

### BEGIN HIDDEN TESTS
assert isinstance(my_factorial(5), numbers.Real)
assert callable(my_factorial)
### END HIDDEN TESTS
```

Exercise 2.5: Compute the length of a path

Some object is moving along a path in the plane. At n points of time, we have recorded the corresponding (x, y) positions of the object: $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$. The total length L of the path from (x_0, y_0) to (x_{n-1}, y_{n-1}) is the sum of all the individual line segments (x_{i-1}, y_{i-1}) to (x_i, y_i), $i = 1, \ldots, n-1$:

$$L = \sum_{i=1}^{n-1}{\sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}}.$$

Create a function `path_length(x, y)` for computing L according to the formula. The arguments `x` and `y` are two lists that hold all the $x_0, \ldots, x_{n-1}$ and $y_0, \ldots, y_{n-1}$ coordinates, respectively. Test the function on a triangular path with the four points (1, 1), (2, 1), (1, 2), and (1, 1).

In [32]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
from math import sqrt
def path_length(x, y):
    L = 0.0
    for i in range(1, len(x)):
        segment = sqrt((x[i] - x[i-1])**2 + (y[i] - y[i-1])**2)
        L += segment
    return L
x = [1, 2, 1, 1]
y = [1, 1, 2, 1]
print(path_length(x,y))
```

```
3.414213562373095
```

In [33]:
```
with pybryt.check(pybryt_reference(2, 5)):
    path_length([-100, 200, -561, 231], [11, 1.1, 2.9, 165.4])
```

```
REFERENCE: exercise-2_5
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! You are adding line segments to the total length.
  - SUCCESS: You set the length to zero before the loop. Well done!
  - SUCCESS: You are computing the length of a single line segment correctly.
  - SUCCESS: Wow! The path length your function returns is correct.
```

In [34]:
```
import numbers
import numpy as np

res = path_length(x=[0, 0, 0, 0, 0], y=[0, 1, 2, 3, 4])
```

```
assert np.isclose(res, 4)

### BEGIN HIDDEN TESTS
res = path_length(x=[0, 0, 0, 0, 0], y=[0, 0, 0, 0, 0])
assert np.isclose(res, 0)
assert isinstance(res, numbers.Real)
assert callable(path_length)
### END HIDDEN TESTS
```

Exercise 2.6: Approximate π

As you know, the circumference of a circle is $2r\pi$ where r is the circle's radius. π is therefore the circumference of a circle with $r= \frac{1}{2}$. We can approximate this circumference by a many-sided polygon through points on the circle. The sum of the lengths of the sides of the polygon will approximate the circumference.

Firstly compute $n+1$ points around a circle according to the formulae:

$ x_i = \frac{1}{2} \cos(\frac{2 \pi i}{n}), y_i = \frac{1}{2} \sin(\frac{2 \pi i}{n}), i = 0 \cdots n$

Then use your `path_length` function from the previous exercise to approximate π. Name your function for estimating π `approx_pi` for testing purposes.

In [35]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
from math import sqrt, cos, sin, pi
def path_length(x, y):
    L = 0.0
    for i in range(1, len(x)):
        segment = sqrt((x[i] - x[i-1])**2 + (y[i] - y[i-1])**2)
        L += segment
    return L
def approx_pi(n):
    x = [1 / 2 * cos(2 * pi * i / n) for i in range(0, n+1)]
    y = [1 / 2 * sin(2 * pi * i / n) for i in range(0, n+1)]
    circumference = path_length(x, y)
    return circumference
print("approx_pi(100) = ",approx_pi(100))
```

```
approx_pi(100) =  3.1410759078128256
```

In [36]:
```
with pybryt.check(pybryt_reference(2, 6)):
    approx_pi(100)
```

```
REFERENCE: exercise-2_6
SATISFIED: True
MESSAGES:
  - SUCCESS: Your computed x-coordinates are correct.
  - SUCCESS: Your computed y-coordinates are correct.
  - SUCCESS: Wow! Your final solution is correct.
```

In [37]:
```
import numbers
import numpy as np

res = approx_pi(800)
assert np.isclose(res, np.pi)

### BEGIN HIDDEN TESTS
assert isinstance(res, numbers.Real)
assert callable(approx_pi)
### END HIDDEN TESTS
```

Exercise 2.7: Make a list of prime numbers

Define a function called `prime_list` that lists all the prime numbers up to a given n.

Hint: Google the *Sieve of Eratosthenes*.

In [38]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.

def prime_list(n):
    primes = []
    for i in range(2, n+1):
        is_prime = True
        for j in range(2, int(i**0.5) + 1):
            if i % j == 0:
                is_prime = False
                break
```

```
        if is_prime:
            primes.append(i)
    return primes

# Example usage:
print(prime_list(100))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

In [39]:

```
with pybryt.check(pybryt_reference(2, 7)):
    prime_list(100)
```

```
REFERENCE: exercise-2_7
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! You are finding primes correctly.
  - SUCCESS: You create an empty list before the loop. Well done!.
  - SUCCESS: Your function returns the right primes.
```

In [40]:

```
import numbers
import numpy as np

res = prime_list(100)
assert isinstance(res, list)
assert len(res) == 25
assert prime_list(0) == []

### BEGIN HIDDEN TESTS
assert callable(prime_list)
assert all([isinstance(i, numbers.Real) for i in res])
assert np.allclose(prime_list(10), [2, 3, 5, 7])
assert prime_list(100) == prime_list(99)
### END HIDDEN TESTS
```

Vectors and arrays

You have known **vectors** since high school mathematics, e.g. point (x, y) in the plane, point (x, y, z) in space. In general, we can describe a vector v as an n-tuple of numbers: $v=(v_0, \ldots, v_{n-1})$. One way to store vectors in Python is by using sequences, e.g. *lists* or *tuples*: v_i is stored as `v[i]`.

Arrays are a generalisation of vectors where we can have multiple indices: A_{ij}, A_{ijk}. In Python, this is represented as a nested list, accessed as `A[i][j]`, `A[i][j][k]`.

Example: Matrices, a table of numbers with one index for the row and one for the column $$ \left\lbrack\begin{array}{cccc} 0 & 12 & -1 & 5q\cr 11 & 5 & 5 & -2 \end{array}\right\rbrack \hspace{1cm} A = \left\lbrack\begin{array}{ccc} A_{0,0} & \cdots & A_{0,n-1}\cr \vdots & \ddots & \vdots\cr A_{m-1,0} & \cdots & A_{m-1,n-1} \end{array}\right\rbrack $$ The number of indices in an array is the *number of dimensions*. Using these terms, a vector can be described as a one-dimensional array or dimension-1 array.

In practice, we use Numerical Python (*NumPy*) arrays instead of lists to represent mathematical arrays because it is **much** faster for large arrays.

Let us consider an example where we store (x,y) points along a curve in Python lists and numpy arrays:

In [41]:
```
# Sample function
def f(x):
    return x**3


# Generate n points in [0, 1]
n = 5
dx = 1 / (n-1)  # x spacing

X = [i*dx for i in range(n)]  # Python list
Y = [f(x) for x in X]

# Turn these Python lists into Numerical Python (NumPy) arrays:
import numpy as np  # as a convention, we import "numpy as np"
```

```
x2 = np.array(X)
y2 = np.array(Y)
```

Instead of first making lists with x and $y = f (x)$ data, and then turning lists into arrays, we can make NumPy arrays directly:

In [42]:
```
n = 5                        # number of points
x2 = np.linspace(0, 1, n)    # generates n points between 0 and 1
y2 = np.zeros(n)
for i in range(n):
    y2[i] = f(x2[i])
```

List comprehensions create lists, not arrays, but we can do:

In [43]:
```
y2 = np.array([f(xi) for xi in x2])  # list -> array
```

Passing a list as an argument to some other function (like `np.array` in this case) is very common. Therefore, Python allows to omit the square brackets `[]`. This results in passing a *generator expression* which is both faster and more memory efficient.

In [44]:
```
y2 = np.array(f(xi) for xi in x2)
```

Since this is not the topic of this introduction to Python lecture series, if you would like to understand more, please refer to PEP289.

When and where to use NumPy arrays

- Python lists can hold any sequence of any Python objects. However, NumPy arrays can only hold objects of the same type. We refer to NumPy arrays as flat sequences, whereas we refer to lists and tuples as containers (or container sequences).
- Arrays are most efficient when the elements are basic number types (*float*, *int*, *complex*).
- In that case, arrays are stored efficiently in the computer's memory, and we can compute very efficiently with the array elements.
- We can compute mathematical operations on whole arrays without loops in Python. For example,

In [45]:
```
x = np.linspace(0, 2, 10001)
y = np.zeros(10001)
for i in range(len(x)):
    y[i] = math.sin(x[i])
```

can be coded as

In [46]:
```
y = np.sin(x)
```

In the latter case, the loop over all elements is now performed in an efficient C-function. Instead of using Python `for`-loops, operations on whole arrays are called vectorisation, and they are a very **convenient**, **efficient**, and therefore an **important** programming technique to master.

Let us consider a simple vectorisation example: a loop to compute x coordinates (`x2`) and $y=f(x)$ coordinates (`y2`) along a function curve:

In [47]:
```
x2 = np.linspace(0, 1, n)
y2 = np.zeros(n)
for i in range(n):
    y2[i] = f(x2[i])
```

This computation can be replaced by:

In [48]:
```
x2 = np.linspace(0, 1, n)
y2 = f(x2)
```

The advantage of this approach is:

- There is no need to allocate space for `y2` (via the NumPy *zeros* function).
- There is no need for a loop.
- It is *much faster*.

How vectorised functions work

Consider the function

In [49]:
```
def f(x):
    return x**3
```

$f(x)$ is intended for a number x, i.e. a *scalar*. So, what happens when we call `f(x2)`, where `x2` is a NumPy array? **The function evaluates x^3 for an array x**. NumPy supports arithmetic operations on arrays, which correspond to the equivalent operations on each element. For example,

In [50]:
```
r1 = x**3                   # x[i]**3 for all i
r2 = np.cos(x)              # cos(x[i]) for all i
r3 = x**3 + x*np.cos(x)     # x[i]**3 + x[i]*cos(x[i]) for all i
r4 = x/3*np.exp(-x*0.5)     # x[i]/3*exp(-x[i]*0.5) for all i
```

In each of these cases, a highly optimised C-function is actually called to evaluate the expression. In this example, the `cos` function called for an `array` is imported from NumPy rather than from the `math` module which only acts on scalars.

Notes:

- Functions that can operate on arrays are called **vectorised functions**.
- Vectorisation is the process of turning a non-vectorised expression/algorithm into a vectorised expression/algorithm.
- Mathematical functions in Python automatically work for both scalar and array (vector) arguments, i.e. no vectorisation is needed by the programmer.

Watch out for references vs. copies of arrays!

Consider this code:

In [51]:
```
a = x
a[-1] = 42
print(x[-1])
```

```
42.0
```

Notice what happened here - we changed a value in `a`, but the corresponding value in `x` was also changed! This is because `a` refers to the same array as `x`. If we want a separate copy of `x`, then we have to make an explicit copy:

In [52]:
```
a = x.copy()
```

We will discuss the references later in lecture 4.

Exercise 2.8: Fill lists and arrays with function values

A function with many applications in science is defined as:

$$h(x) = \frac{1}{\sqrt{2\pi}}\text{e}^{-\frac{1}{2}x^2}$$

- Implement the above formula as a Python function. Call the function `h` and it should take just one argument, `x`.
- Create a NumPy array (call it `x`) that has 9 uniformly spaced points in $[-4, 4]$.
- Create a second NumPy array (call it `y`) with the function `h(x)`.

In [53]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
import numpy as np
def h(x):
    h_x = np.exp(-0.5 * x**2) / np.sqrt(2*np.pi)
    return h_x

x = np.linspace(-4, 4, 9)
y = np.zeros(9)
for i in range(9):
    y[i] = h(x[i])
```

In [54]:
```
with pybryt.check(pybryt_reference(2, 8)):
    h(5), x, y
```

```
REFERENCE: exercise-2_8
SATISFIED: True
MESSAGES:
  - SUCCESS: Your function computes h(x)c orrectly. Well done!
  - SUCCESS: Great! You created array x correctly.
  - SUCCESS: Great! You created array y correctly.
```

In [55]:
```
import numbers
import numpy as np

assert isinstance(h(10), numbers.Real)
assert np.isclose(h(0), 1/np.sqrt(2*np.pi))
assert x.shape == y.shape == (9,)
assert np.isclose(x[0], -4)
assert np.isclose(x[-1], 4)

### BEGIN HIDDEN TESTS
assert callable(h)
assert all([isinstance(i, np.ndarray) for i in [x, y]])

for i in np.linspace(-100, 100, 51):
    assert isinstance(h(i), numbers.Real)
### END HIDDEN TESTS
```

Generalised array indexing

We can select a slice of an array using `a[start:stop:inc]`, where the slice `start:stop:inc` implies a set of indices starting from `start`, up to `stop` in increments `inc`. Any integer list or array can be used to indicate a set of indices:

In [56]:
```
a = np.linspace(1, 8, 8)
print(a)
```

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

In [57]:
```
a[[1, 6, 7]] = 10  # i.e. set the elements with indicies 1, 6, and 7 in the array to 10.
print(a)
```

```
[ 1. 10.  3.  4.  5.  6. 10. 10.]
```

In [58]:
```
a[range(2, 8, 3)] = -2   # same as a[2:8:3] = -2
print(a)
```

```
[ 1. 10. -2.  4.  5. -2. 10. 10.]
```

Even boolean expressions can be used to select part of an array(!)

In [59]:
```
print(a[a < 0])  # pick out all negative elements
```

```
[-2. -2.]
```

In [60]:
```
a[a < 0] = a.max()  # if a[i]<0, set a[i]=10
print(a)
```

```
[ 1. 10. 10.  4.  5. 10. 10. 10.]
```

Exercise 2.9: Explore array slicing

- Create a NumPy array called `w` with 31 uniformly spaced values ranging from 0 to 3.
- Using array slicing, create a NumPy array called `wbits` that starts from the 4^{th} element of `w`, excludes the final element of `w` and selects every 3^{rd} element.

In [61]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
import numpy as np
w = np.linspace(0, 3, 31)
wbits = w[3:-1:3]
print(wbits)
```

```
[0.3 0.6 0.9 1.2 1.5 1.8 2.1 2.4 2.7]
```

In [62]:
```
with pybryt.check(pybryt_reference(2, 9)):
    w, wbits
```

```
REFERENCE: exercise-2_9
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! You created array w correctly.
  - SUCCESS: You sliced array w correctly. Amazing!
```

In [63]:
```
import numbers
import numpy as np

assert all(isinstance(i, np.ndarray) for i in [w, wbits])
assert w.shape == (31,)
assert wbits.shape == (9,)
assert np.isclose(w[-1] - w[0], 3)

### BEGIN HIDDEN TESTS
assert np.allclose(wbits, [0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7])
### END HIDDEN TESTS
```

2D arrays

When we have a table of numbers,

$$ \left\lbrack\begin{array}{cccc} 0 & 12 & -1 & 5\cr -1 & -1 & -1 & 0\cr 11 & 5 & 5 & -2 \end{array}\right\rbrack $$

(i.e. a *matrix*) it is natural to use a two-dimensional array $A_{i, j}$ with one index for the rows and one for the columns:

$$ A = \left\lbrack\begin{array}{ccc} A_{0,0} & \cdots & A_{0,n-1}\cr \vdots & \ddots & \vdots\cr A_{m-1,0} & \cdots & A_{m-1,n-1} \end{array}\right\rbrack $$

Let us recreate this array using NumPy:

In [64]:
```
A = np.zeros((3, 4))  # we create a 2-dimensional (3 x 4) array filled with zeros

A[0, 0] = 0
A[1, 0] = -1
A[2, 0] = 11

A[0, 1] = 12
A[1, 1] = -1
A[2, 1] = 5

A[0, 2] = -1
A[1, 2] = -1
```

```
A[2, 2] = 5

# we can also use the same syntax that we used for nested lists

A[0][3] = 5
A[1][3] = 0
A[2][3] = -2

print(A)
```

```
[[ 0. 12. -1.  5.]
 [-1. -1. -1.  0.]
 [11.  5.  5. -2.]]
```

Next, let us create a nested list and then convert into a 2D array:

In [65]:
```
Cdegrees = range(0, 101, 10)
Fdegrees = [9/5*C + 32 for C in Cdegrees]
table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]  # create a nested list
print(table)
```

```
[[0, 32.0], [10, 50.0], [20, 68.0], [30, 86.0], [40, 104.0], [50, 122.0], [60, 140.0], [70, 158.0], [80, 176.0], [90, 194.0],
[100, 212.0]]
```

In [66]:
```
# Convert this nested list into a NumPy array:
table2 = np.array(table)
print(table2)
```

```
[[  0.  32.]
 [ 10.  50.]
 [ 20.  68.]
 [ 30.  86.]
 [ 40. 104.]
 [ 50. 122.]
 [ 60. 140.]
 [ 70. 158.]
 [ 80. 176.]
 [ 90. 194.]
 [100. 212.]]
```

To see the number of elements in each dimension we ask for array's `shape`:

In [67]:
```
print(table2.shape)
```

```
(11, 2)
```

i.e. our table has 11 rows and 2 columns.

Let us write a loop over all array elements of A:

In [68]:
```
for i in range(table2.shape[0]):
    for j in range(table2.shape[1]):
        print(f"table2[{i}, {j}] = {table2[i, j]}")
```

```
table2[0, 0] = 0.0
table2[0, 1] = 32.0
table2[1, 0] = 10.0
table2[1, 1] = 50.0
table2[2, 0] = 20.0
table2[2, 1] = 68.0
table2[3, 0] = 30.0
table2[3, 1] = 86.0
table2[4, 0] = 40.0
table2[4, 1] = 104.0
table2[5, 0] = 50.0
table2[5, 1] = 122.0
table2[6, 0] = 60.0
table2[6, 1] = 140.0
table2[7, 0] = 70.0
table2[7, 1] = 158.0
table2[8, 0] = 80.0
table2[8, 1] = 176.0
table2[9, 0] = 90.0
table2[9, 1] = 194.0
table2[10, 0] = 100.0
table2[10, 1] = 212.0
```

Alternatively:

In [69]:
```
for index_tuple, value in np.ndenumerate(table2):
    print(f"index {index_tuple} has value {value}")
```

```
index (0, 0) has value 0.0
index (0, 1) has value 32.0
index (1, 0) has value 10.0
index (1, 1) has value 50.0
index (2, 0) has value 20.0
index (2, 1) has value 68.0
index (3, 0) has value 30.0
index (3, 1) has value 86.0
index (4, 0) has value 40.0
index (4, 1) has value 104.0
index (5, 0) has value 50.0
index (5, 1) has value 122.0
index (6, 0) has value 60.0
index (6, 1) has value 140.0
index (7, 0) has value 70.0
index (7, 1) has value 158.0
index (8, 0) has value 80.0
index (8, 1) has value 176.0
index (9, 0) has value 90.0
index (9, 1) has value 194.0
index (10, 0) has value 100.0
index (10, 1) has value 212.0
```

We can also extract slices from multi-dimensional arrays as before. For example, extract the second column:

In [70]:
```
print(table2[:, 1])  # 2nd column (index 1)
```

```
[ 32.  50.  68.  86. 104. 122. 140. 158. 176. 194. 212.]
```

Play with this more complicated example:

In [71]:
```
t = np.linspace(1, 30, 30).reshape(5, 6)
print(t)
```

```
[[ 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.]]
```

In [72]:
```
print(t[1:-1:2, 2:])
```

```
[[ 9. 10. 11. 12.]
 [21. 22. 23. 24.]]
```

Exercise 2.10: Matrix-vector multiplication

A matrix \$\mathbf{A}\$ and a vector \$\mathbf{b}\$, represented in Python as a 2D array and a 1D array, respectively, are given by:

\$\$ \mathbf{A} = \left\lbrack\begin{array}{ccc} 0 & 12 & -1\cr -1 & -1 & -1\cr 11 & 5 & 5 \end{array}\right\rbrack \$\$

\$\$ \mathbf{b} = \left\lbrack\begin{array}{c} -2\cr 1\cr 7 \end{array}\right\rbrack \$\$

Multiplying a matrix by a vector results in another vector \$\mathbf{c}\$, whose components are defined by the general rule:

\$\$\mathbf{c}_i = \sum_j\mathbf{A}_{i, j}\mathbf{b}_j\$\$

- Define \$\mathbf{A}\$ and \$\mathbf{b}\$ as NumPy arrays
- Write a function called `multiply` that takes two arguments, a matrix and a vector in the form of NumPy arrays, and returns a NumPy array containing their product.
- Call this function on \$\mathbf{A}\$ and \$\mathbf{b}\$, and store the result in a variable \$c\$.

In [73]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
import numpy as np
def multiply(A, b):
    r, c = A.shape
    result = np.zeros(r)

    for i in range(r):
        for j in range(c):
            result[i] += A[i, j] * b[j]

    return result
A = np.array([[0, 12, -1],
              [-1, -1, -1],
              [11, 5, 5]])
```

```
b = np.array([-2, 1, 7])

c = multiply(A, b)
print(c)
```

```
[ 5. -6. 18.]
```

In [74]:
```
with pybryt.check(pybryt_reference(2, 10)):
    A, b, c, multiply(A, b)
```

```
REFERENCE: exercise-2_10
SATISFIED: True
MESSAGES:
  - SUCCESS: Your matrix A is correct. Nice!
  - SUCCESS: Your vector b is correct. Nice!
  - SUCCESS: You are populating resulting vector correctly. Amazing!
  - SUCCESS: Great! You created a zero-vector before the loop.
  - SUCCESS: Wow! The result of your multiplication is correct.
```

In [75]:
```
import numbers
import numpy as np

assert all([isinstance(i, np.ndarray) for i in [A, b, c]])
assert b.shape == c.shape
assert A.shape == (3, 3)
assert np.allclose(multiply(A, b), c)
assert np.allclose(multiply(np.identity(3), np.array([5, 9, -11.1])), [5, 9, -11.1])

### BEGIN HIDDEN TESTS
assert callable(multiply)

assert np.allclose(c, [5, -6, 18])

assert all([isinstance(i, numbers.Real) for i in A.flatten()])
assert all([isinstance(i, numbers.Real) for i in b])
assert all([isinstance(i, numbers.Real) for i in c])

assert np.allclose(multiply(np.array([[0]]), np.array([0])), [0])
### END HIDDEN TESTS
```

`@` operator

Matrix-matrix or matrix-vector multiplication is a very common operation in computational and data science. Therefore, NumPy provides a convenience operator `@` that can be applied between arrays of compatible shapes for multiplication. For example,

In [76]:
```
A = np.array([[1, 1], [2, 1]])
b = np.array([1, 2])
Ab = A @ b

print(f"{A = }")
print(f"{b = }")
print(f"{Ab = }")
```

```
A = array([[1, 1],
       [2, 1]])
b = array([1, 2])
Ab = array([3, 4])
```

Exercise 2.11: Vectorised function

Let A_{33} be the two-dimensional array

$$ \mathbf{A_{33}} = \left\lbrack\begin{array}{ccc} 0 & 12 & -1\cr -1 & -1 & -1\cr 11 & 5 & 5 \end{array}\right\rbrack $$

Implement and apply the function

$$f(x) = x^3 + xe^x + 1$$

to each element in A_{33}. Then calculate the result of the array expression ${A_{33}}^3 + A_{33}e^{A_{33}} + 1$, and demonstrate that the end result of the two methods are the same.

In [77]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
import numpy as np
A33 = np.array([[0, 12, -1],
             [-1, -1, -1],
```

```
              [11, 5, 5]])

def f_cubic(A):
    f_x = A**3 + A * np.exp(A) + 1
    return f_x

result1 = f_cubic(A)


result2 = np.zeros_like(A, dtype=float)
rows, cols = A.shape

for i in range(rows):
    for j in range(cols):
        x = A[i, j]
        result2[i, j] = x**3 + x * np.exp(x) + 1

print(result1)
print(result2)
print("\nIs the result1 the same with result2?", np.allclose(result1, result2))
```

```
[[ 4.71828183  4.71828183]
 [23.7781122   4.71828183]]
[[ 4.71828183  4.71828183]
 [23.7781122   4.71828183]]

Is the result1 the same with result2? True
```

In [78]:

```
with pybryt.check(pybryt_reference(2, 11)):
    f_cubic(np.array([[1, 2, -6], [2, 2, -5]])), A33
```

```
REFERENCE: exercise-2_11
SATISFIED: True
MESSAGES:
  - SUCCESS: Wow! The result of your function is correct.
  - SUCCESS: Your matrix A33 is correct.
```

In [79]:

```
import numbers
import numpy as np

res = f_cubic(A33)
```

```
assert isinstance(A33, np.ndarray)
assert all([isinstance(i, numbers.Real) for i in A33.flatten()])
assert A33.shape == f_cubic(A33).shape == (3, 3)
assert np.isclose(np.sum(A33), 29)

### BEGIN HIDDEN TESTS
assert callable(f_cubic)
assert np.allclose(f_cubic(A33), np.power(A33, 3) + np.multiply(A33, np.exp(A33)) + 1)
assert all([isinstance(i, numbers.Real) for i in res.flatten()])
assert np.allclose(f_cubic(np.array([0])), 1)
### END HIDDEN TESTS
```

Exercise 2.12: Matrix-matrix multiplication

The general rule for multiplying an $n \times m$ matrix $\mathbf{A}$ by a $m \times p$ matrix $\mathbf{B}$ results in a $n \times p$ matrix $\mathbf{C}$, whose components are defined by the general rule

$$\mathbf{C}_{i,j} = \sum^m_{k=1}\mathbf{A}_{i,k}\mathbf{B}_{k,j}$$

Again let $\mathbf{A}$ be the two-dimensional array

$$ \mathbf{A} = \left\lbrack\begin{array}{ccc} 0 & 12 & -1\cr -1 & -1 & -1\cr 11 & 5 & 5 \end{array}\right\rbrack $$

and let $\mathbf{B}$ be the two-dimensional array

$$ \mathbf{B} = \left\lbrack\begin{array}{ccc} -2 & 1 & 7\cr 3 & 0 & 6\cr 2 & 3 & 5 \end{array}\right\rbrack. $$

Define `A` and `B` as NumPy arrays, and write a function `f_mult` which multiplies them together using the above rule. Save the result of multiplication `f_mult(A, B)` in variable `C`.

In [80]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
import numpy as np
A = np.array([[0, 12, -1],
              [-1, -1, -1],
              [11, 5, 5]])

B = np.array([[-2, 1, 7],
```

```
              [3, 0, 6],
              [2, 3, 5]])

def f_mult(A, B):
    if A.shape[1] != B.shape[0]:
        raise RuntimeError('Matrix A should have the same number of columns as B has rows.')

    res = np.zeros([A.shape[0], B.shape[1]])
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            for k in range(A.shape[1]):
                res[i, j] += A[i, k] * B[k, j]

    return res

C = f_mult(A, B)
```

In [81]:

```
with pybryt.check(pybryt_reference(2, 12)):
    f_mult(np.array([[5, 6], [11, 17]]), np.array([[91, -6], [21, 14]])), A, B
```

```
REFERENCE: exercise-2_12
SATISFIED: True
MESSAGES:
  - SUCCESS: You are populating resulting vector correctly. Amazing!
  - SUCCESS: You created a zero-matrix before looping.
  - SUCCESS: Amazing! Your function computes the multiplication correctly.
  - SUCCESS: Your matrix A is correct.
  - SUCCESS: Your matrix B is correct.
```

In [82]:

```
import numbers
import numpy as np

assert all([isinstance(i, np.ndarray) for i in [A, B, C]])
assert A.shape == B.shape == C.shape == (3, 3)
assert np.isclose(np.sum(A), 29)
assert np.isclose(np.sum(B), 25)

assert all([isinstance(i, numbers.Real) for i in A.flatten()])
assert all([isinstance(i, numbers.Real) for i in B.flatten()])
assert all([isinstance(i, numbers.Real) for i in C.flatten()])
```

```
assert A33.shape == f_cubic(A33).shape == (3, 3)
assert np.isclose(np.sum(A33), 29)

### BEGIN HIDDEN TESTS
assert callable(f_mult)
assert np.allclose(C, np.array([[34, -3, 67], [-3, -4, -18], [3, 26, 132]]))
assert np.allclose(C, f_mult(A, B))
assert np.allclose(f_mult(np.zeros((3, 3)), np.zeros((3, 3))), 0)
assert np.allclose(f_mult(np.identity(10), np.identity(10)), np.identity(10))
### END HIDDEN TESTS
```

Exercise 2.13: 2D array slicing

- Create a 1D NumPy array called `odd` with all of the odd numbers from 1 to 55
- Create a 2D NumPy array called `odd_sq` with all of the odd numbers from 1 to 55 in a matrix with 4 rows and 7 columns
- Using array slicing, create a 2D NumPy array called, `odd_bits`, that starts from the 2^{nd} column of `odd_sq` and selects every other column, of only the 2^{nd} and 3^{rd} rows of `odd_sq`.

In [83]:
```
# Uncomment and complete this code - keep the names the same for testing purposes.
import numpy as np

# step 1
odd = np.arange(1, 56, 2)
print(odd)

# step 2
odd_sq = np.arange(1, 56, 2).reshape(4, 7)
print(odd_sq)

# step 3
odd_bits = odd_sq[1:3, 1::2]
print(odd_bits)
```

```
[ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47
 49 51 53 55]
[[ 1  3  5  7  9 11 13]
 [15 17 19 21 23 25 27]
 [29 31 33 35 37 39 41]
 [43 45 47 49 51 53 55]]
[[17 21 25]
 [31 35 39]]
```

In [84]:

```python
with pybryt.check(pybryt_reference(2, 13)):
    odd, odd_sq, odd_bits
```

```
REFERENCE: exercise-2_13
SATISFIED: True
MESSAGES:
  - SUCCESS: Great! The array odd is correct.
  - SUCCESS: You reshaped the array correctly.
  - SUCCESS: Wow! You sliced the array correctly.
```

In [85]:

```python
import numbers
import numpy as np

assert all(isinstance(i, np.ndarray) for i in [odd, odd_sq, odd_bits])
assert odd.shape == (28,)
assert odd_sq.shape == (4, 7)
assert odd_bits.shape == (2, 3)
assert all([i % 2 for i in odd])

### BEGIN HIDDEN TESTS
res = np.arange(1, 56, 2)
assert np.allclose(odd, np.arange(1, 56, 2))
assert np.allclose(odd_sq, np.arange(1, 56, 2).reshape(4, 7))
assert np.allclose(odd_bits, np.arange(1, 56, 2).reshape(4, 7)[1:3, 1::2])
### END HIDDEN TESTS
```

In [ ]: